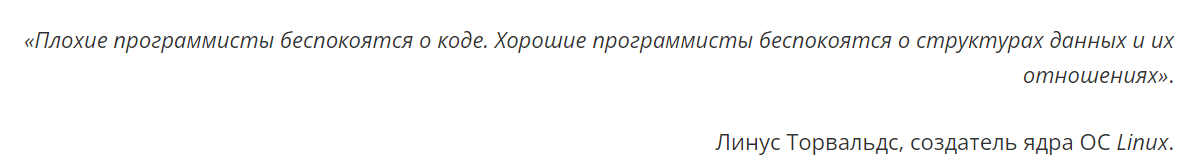
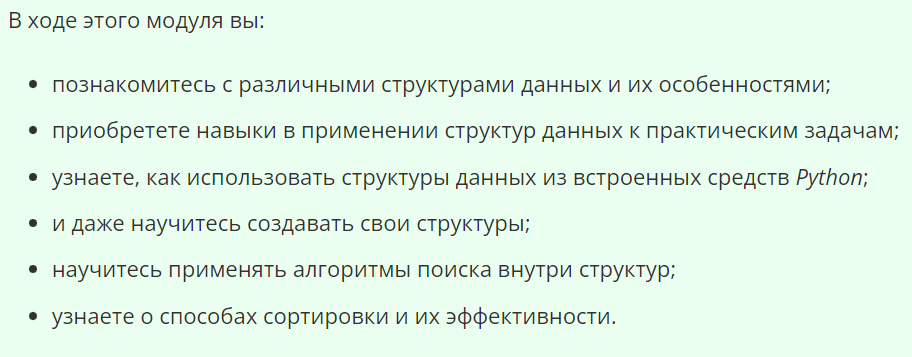
**Модуль B9. Алгоритмы и структуры данных**



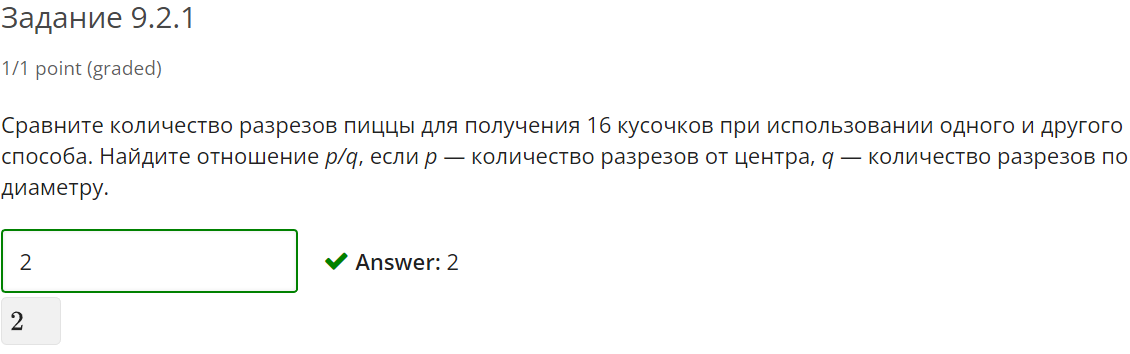
Да, да. Может быть когда-нибудь. Чую этот модуль будет ну\*(д/ж)ным.



Но к этому лучше в другой день. И снова этот день настал.

Пример с пиццей, можно разрезать ее от центра. И тогда количество разрезов будет эквивалентно количеству кусков пиццы: 0 разрезов - 1 целая, 1 разрез – 1 надрезанная, 2 разреза, 1 кусок и 1 остаток пиццы.

А можно резать ее по диаметру до конца и сразу получить четное количество кусков. Я реально это пишу? Да.. этого даже задача касалась.



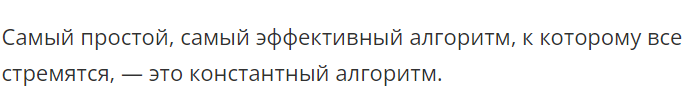


Сложность алгоритмов… да, что-то такое было.

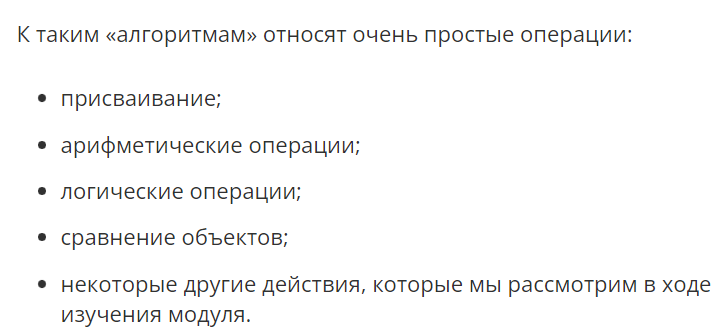
Существует **временная** и **пространственная** сложность.

**Временная сложность** связана напрямую с тем, сколько алгоритм будет выполняться. Может зависеть от количество обрабатываемых данных.

**Пространственную сложность** затрагиваютреже. Она касается напрямую затрагиваемой памяти. Чем больше переменных, мб лишних действий и тд будет в алгоритме, тем больше памяти он потребует.



Впервые слышу. **Константный** или **постоянный по времени** это алгоритм который выполняет необходимое действие за всегда одинаковое количество времени.



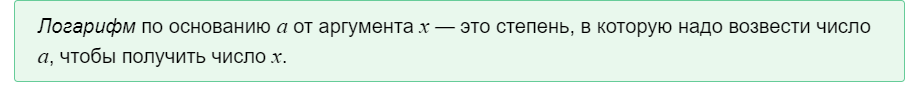
Более сложные, например найти наибольший элемент в списке уже сложнее. Будет зависеть от количества данных, 3 штуки или 3 миллиона? Самым простым вариантом будет пройтись по всем элементам списка. Время вычисления будет расти прямо-пропорционально данным. Такой алгоритм будет считаться **линейным**.

Бывают ситуации, когда алгоритмы могут работать быстрее чем **линейные**. Самые распространенные – это **логарифмические** **алгоритмы**. Блин, уже не помню почти ничего о логарифмах. Придется вспомнить.

Их название напрямую связано с тем, как возрастает временная сложность алгоритма с ростом размера входных данных.

Принято считать основанием логарифма двойку если не указано иное.

Вспоминаем логарифмы!!

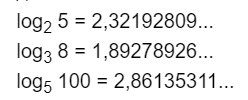






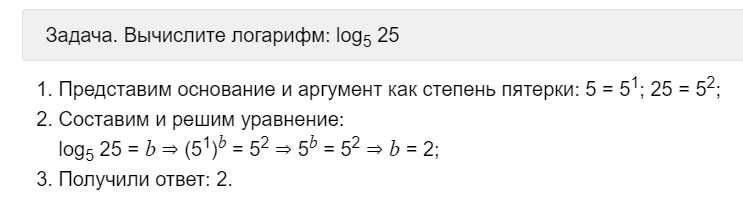
Угу, ага. А зачем?

Но это простые логарифмы. А вот те, которые не получаются точно.



**Эти цифры напомню называются иррациональными.** Те, у которых можно писать цифры после запятой и они никогда не повторяться. Ясно. Значит делители числа 7 другие. Они повторяются.

Аргумент и основание. Всегда должны быть больше 0.





Ну да, рационального ответа нет, но что насчет 7^1.356 = Это почти 14.

Также существует десятичные логорифмы. Это логорифмы по основанию 10. Пишутся просто lg.

Мб потому что **g** это отсылка к числу **g** скорости свободного падения или уже не помню на нашей планете, которая почти равна 10.

Кстати. Как вычислять логорифм на калькуляторе. Да, я не школьник мне уже можно.

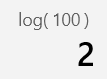


Надо отдельно логарифм верхнего числа, поделить на логарифм нижнего числа.



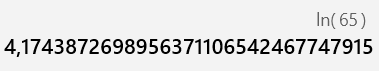
А теперь объясните мне, а как самому получить логарифм 8? Я имею ввиду какое основание используется 10?

Похоже, что да?

 10^x == 100 == 10^2. x = 2

Ну и на последок проверю ln. Насколько я знаю это логарифм числа по основанию 2.7

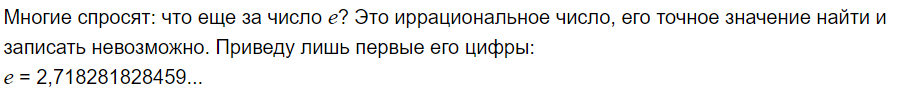
ln 65 = 4.20274?



LN это не 2.7 может 2.07. Ща…

**2.7 это e**





Как и число Пи. А это случайно не цифры числа 7? Не… У 7 такие: **142857**

**Так…**

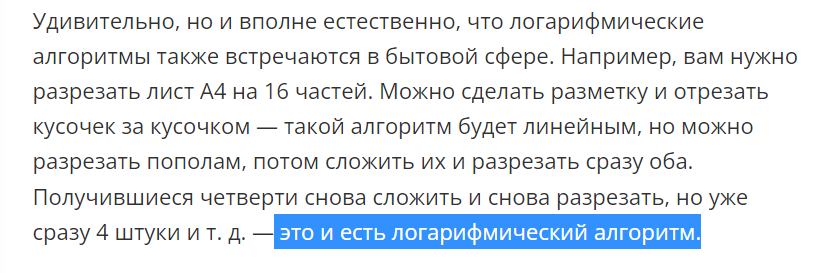
**Но это все же ln был.**

**Понятно. Я ошибся потому что использовал только часть основания 2.7, а калькулятор использовал более точное 2.718… и тд. Для моего числа я подобрал максимально близкое.**

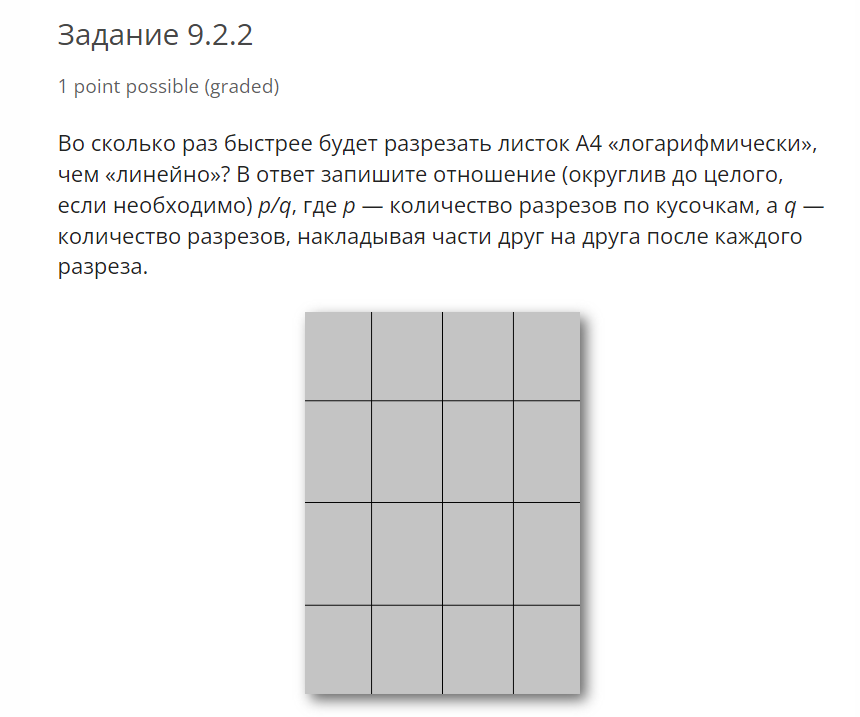
Если внутри ln используется число e. То ответом логарифма будет степень числа e. Потому что в основании и так e.



Ладно. Вернемся к программированию.



**Это.. довольно понятный пример.**





Линейно 24 разреза. А логарифмически 4 разреза. Ответ 24/4 = 6

Чтобы раставить 10 книг на полке в алфавитном порядке нужно провести такое количество сравнений:



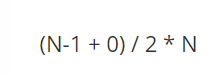
45. Если бы в этой арифметической прогрессии было бы миллион чисел, было бы сложнее выяснить результат. Для этого есть формула:

**найти среднее арифметическое первого и последнего членов прогрессии и умножить на количество всех членов.**

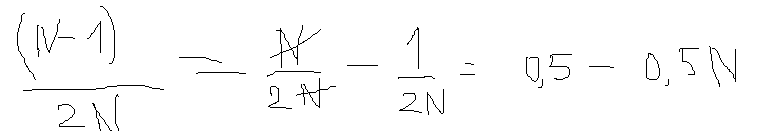
Попробую для такой же прогрессии но из миллиона чисел.

(999999 + 0) / 2 \* 1000000 = 499 999 500 000

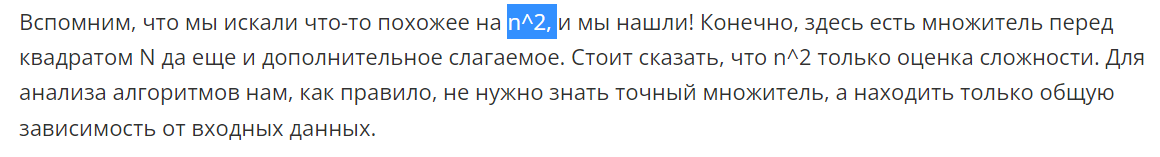
Таким алгоритмом можно посчитать такую прогрессию.



Сразу как-то не понял как произошло это упрощение. Нука…



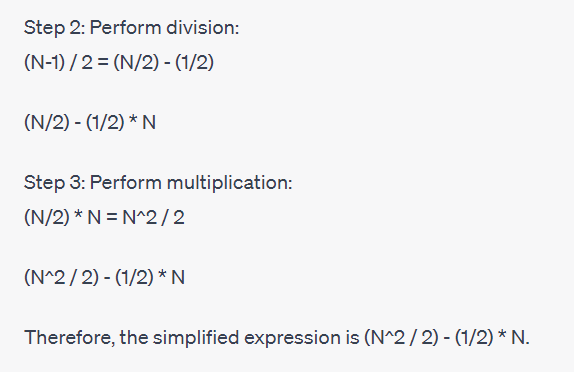
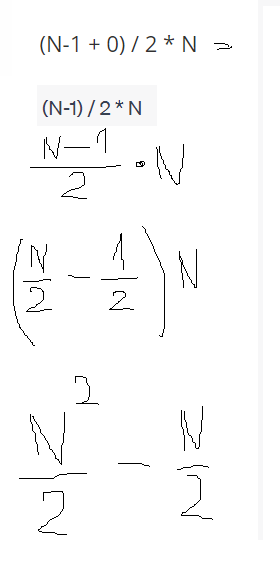
Да! Нахер там квадрат. Чтобы было похоже на линейное уравнение? x^2 + ax + b



Ясно, все для этого? Эм.. честно не понял, мб я считать разучился, возможно. Они просто выражение умножили еще раз на **N! Это не упрощение, а уложнение. Как они получили это число!!!**



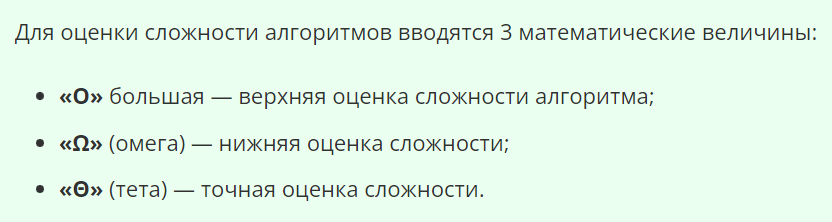
**Долбоебы!!!** Но как и я полагаю. GPT такой же ответ выдал, посмотрю как.

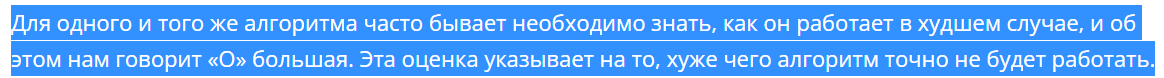
****

**Это я тупой(((**

**простите skillFactory.**

Случайно объединил операцию деления на 2 и N. Получилось будто полученное выражение я лишил одной степени N.

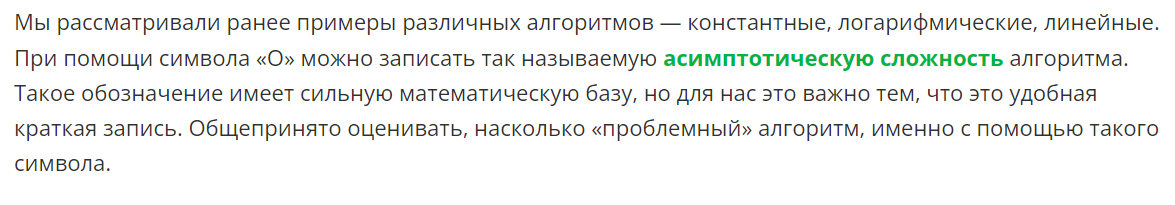


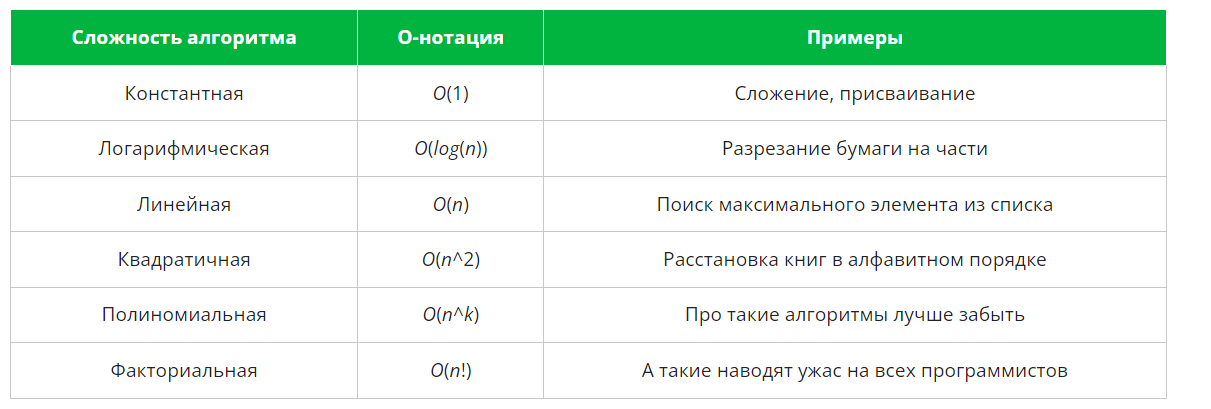


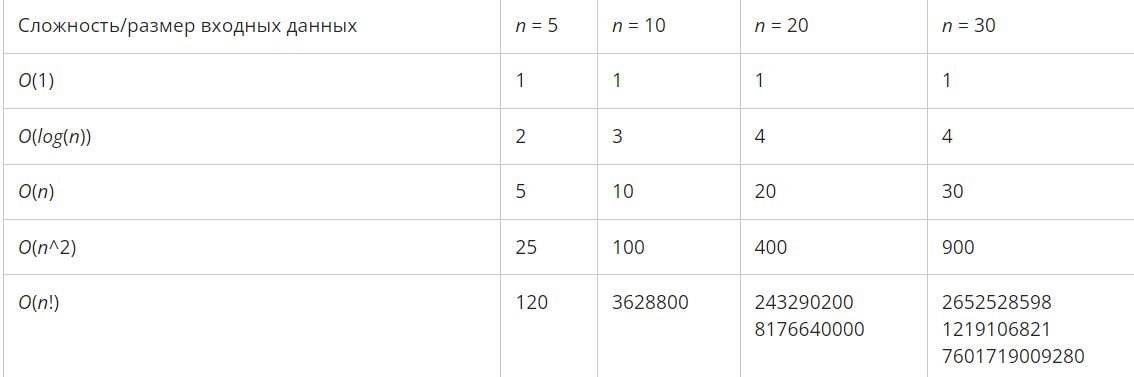
Не понял.

Это типо максимально сложная работа алгоритма сложнее которой он уже не отработает?

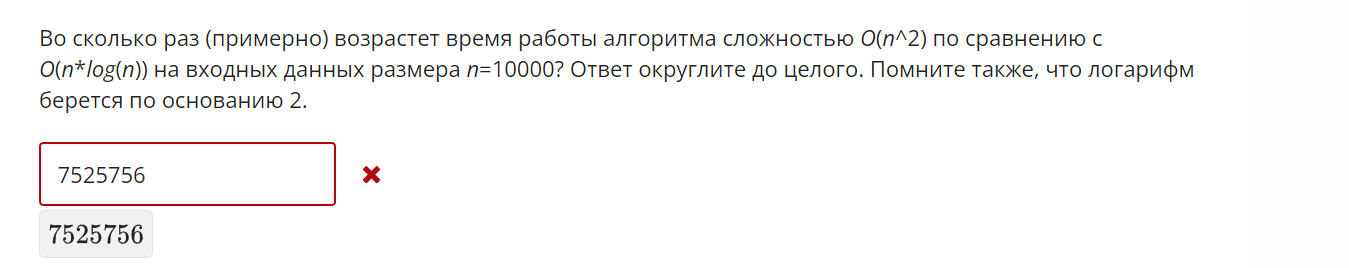
Тут видимо да речь он входных данных. Максимально плохих. Там.. террабайты данных или типо того…   
Нижняя оценка сложности, это работа алгоритма в идеальных условиях. Ну например при игре в морской бой компьютер внезапно с первого раза угадает все корабли.







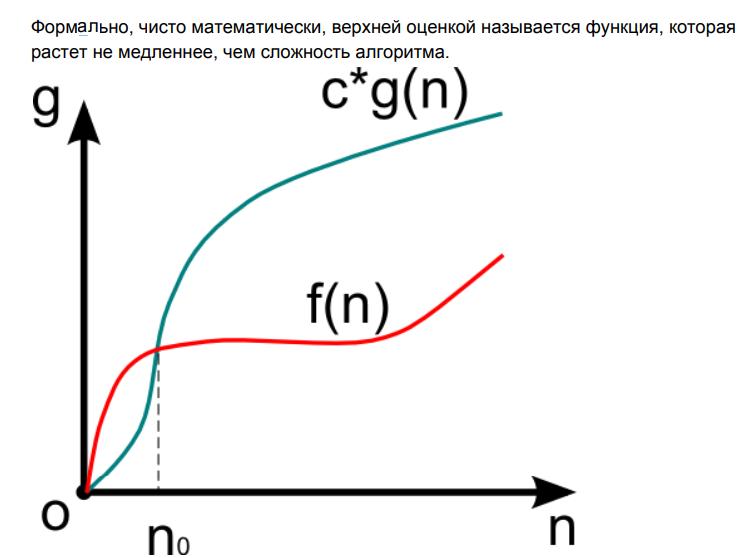
Для логарифма log2 30 = не 4, а 4.9

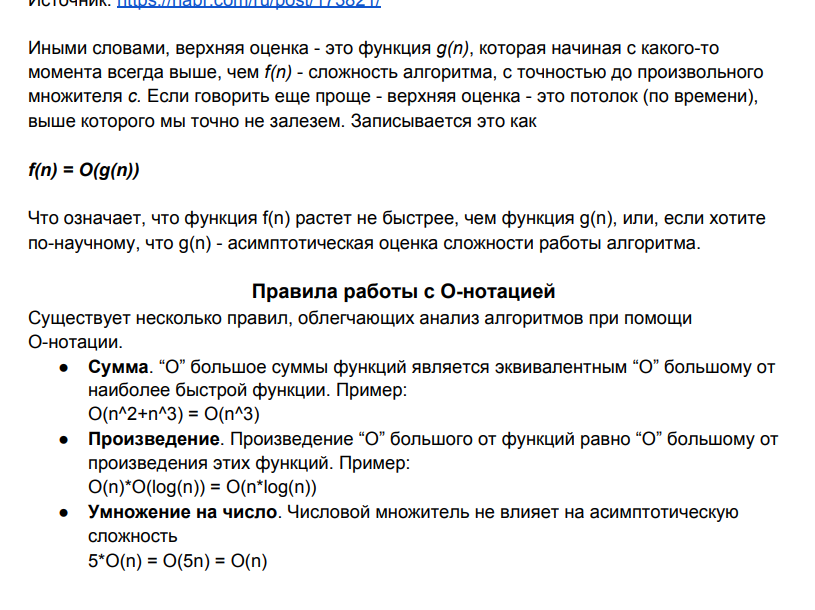


Ладно. В чем ошибка? Округлить вверх? Нет. А забыл на 10000 логорифм умножить.

132877.123 и 100000000. Ответ просто 752? **\*753**

Еее.. еще могу!!! (с калькулятором… ну и ладно, проги собственно для этого и пишут, чтобы самим не решать)



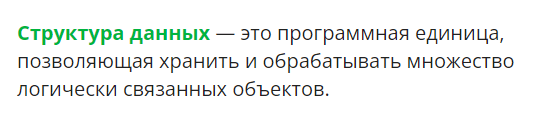


Ну правила понятны. Хотя их стоит повторять.

**B9.3. Основные структуры данных**

Нужно сделать перерыв, а то я сегодня совсем не работал. А сегодня я чуть поработал, а еще читал мангу. Еще чуть почитаю, и продолжу… Или нет. Ладно. Сегодня я и поработал и почитал и еще не поздно. Позанимаюсь.

Бла, бла, бла. Структуры данных. Свои классы, которые внутри себя могут содержать различные формы данных.

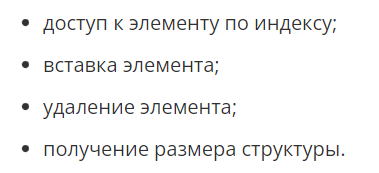


Сначала рассмотрим такие структуры данных как **массивы** и **Хэштаблицы.**

**Массивы** понятное дело, бывают статические и динамические. К структуре данных массива в **Python** относят списки. **Хэштаблицы** не уверен, но полагаю имеются ввиду что-то вроде словарей и тд.

Свои мысли закончил, а теперь что на деле…

Рассматривать структуры данных будем с точки зрения ее хранения в памяти, и способов взаимодействия с этой структурой. Например:



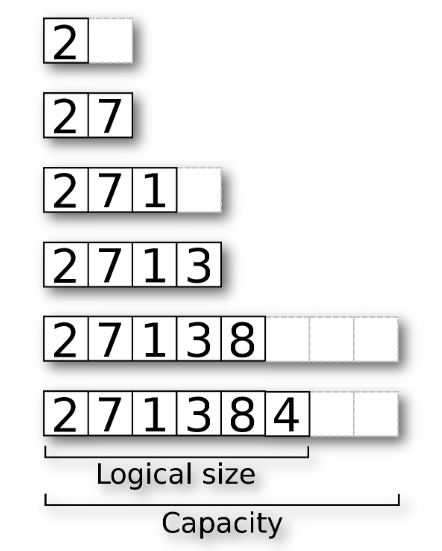
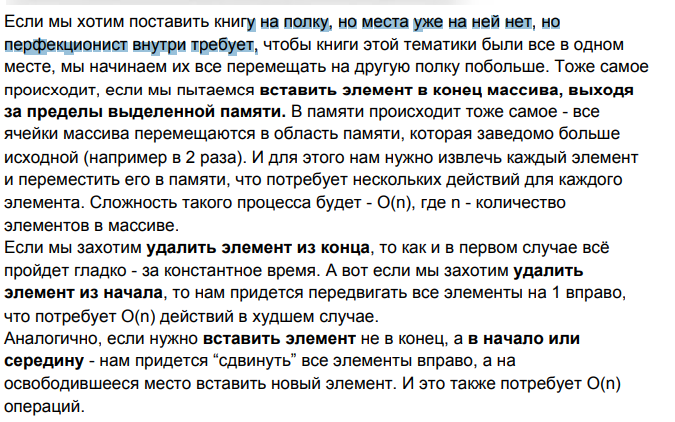
Если бы меня спросили что такое массив, я бы объяснил. Но термин не помню. А так,

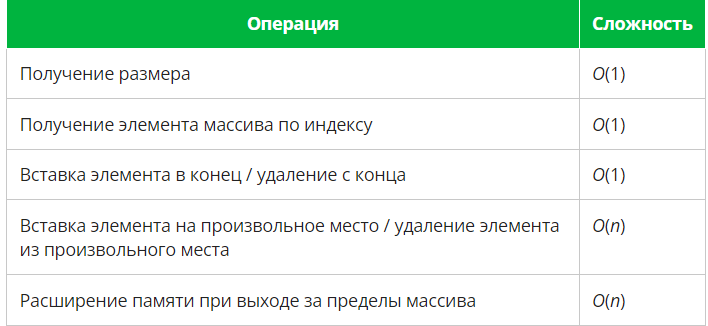
**массив –** это упорядоченный набор данных. Важно не путать упорядоченные структуры и отсортированные, о них тоже будет. Массивы бывают одномерных, двумерные и тд…

Особенность в том, что они хранят элементы в последовательных ячейках памяти. При его создании всегда нужно указывать какое количество памяти нужно занять.

Такие массивы называются **статическими**, помню в плюсах были.

Помимо них существуют **динамические массивы**. Они с помощью буферного механизма могут изменять свой изначальный размер. Важно не путать логический размер. То, насколько массив сейчас чем-то заполнен и **Capacity,** размер массива в памяти.



**List** в **python** не массив, а **динамический массив.** Да вы что…



На этом документе пока остановлюсь.

Какой интересный код. Есть модуль, который замеряет Сколько будет исполнятся тот или иной код.

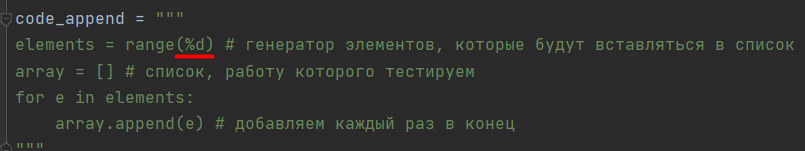
Модуль **timeit,** а функция:



Тоже **timeit.** В качестве первого аргумента она принимает функцию. Но сразу вопрос что это за процент **size**?

Сейчас объясню. Через него мы будем передавать в функцию **range**.

Сама функция должна быть передана в форме строки с корректными отступами. Потому что этот код будет действительно отработан во время проверки. Вот пример такой строки:



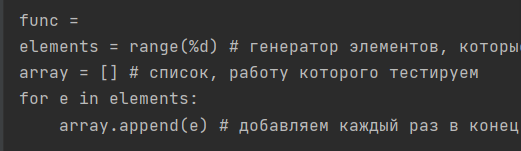
Элементарная код, которая создает список из **%d** чисел. И затем добавляет из этого списка эти числа в другой список через **append.** Именно для того, чтобы этот код можно было проверять этот код с разными аргументами и используется процент. Это схоже с **f**строками. Сначала идет строка внутри которой есть разные аргументы с процентами, а затем через % эти аргументы записываются по порядку. Вот простой пример из этой же программы, как такие проценты пишутся.



Только скажу честно не сильно практиковал. Но число **ratio,** оставляет только 2 знака после запятой. Но, что значит 5 с точкой я не помню.

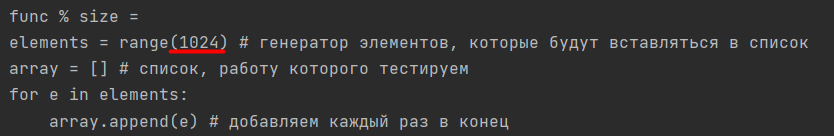
Так вот, если переменную func, попытаться вывести без size будет так:



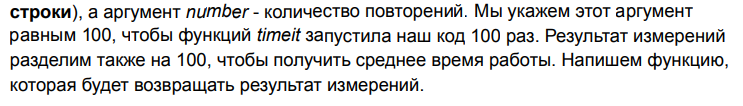


А если передать:





А второй аргумент в функции **timeit.timeit –** это **number.**



В итоге запусти эту функцию с добавляением:

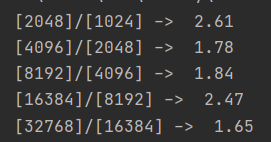
Получились такие примеры, так как время крайне мало, проще будет смотреть с числом больше 1. То произвели деление выполнения этой функции степени n + 1 / степени n



5 циклов от числа s 10, до 15.

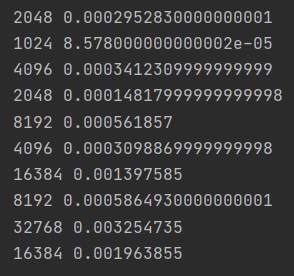


Результаты по времени:



Сейчас, прочитаю объяснение. Просто не знаю это время или память или хз что. Но чем больше код выполнялся, тем меньше результат. Если проводить обычное деление, то ответ будет 2. Но тут показатели range, в функции добавления.

Если рассматривать их отдельно без деления, то вот:

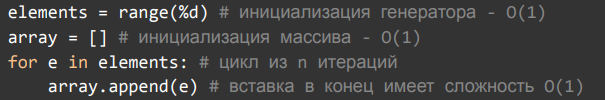


Не обманитесь числом 1024. Там есть е-05, Что значит чисто меньше в 10000 раз.

Так что говорят результаты:

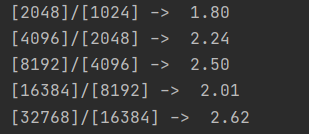
Результаты говорят, что с ростом размера массива, время выполнения увеличивается примерно в 2 раза. В таком случае сложность равна:





Посмотрим на ситуацию вставки элемента в начало массива:

Для метода **insert(0, e).** Результаты по мере роста длины массива от 2, до 2.5



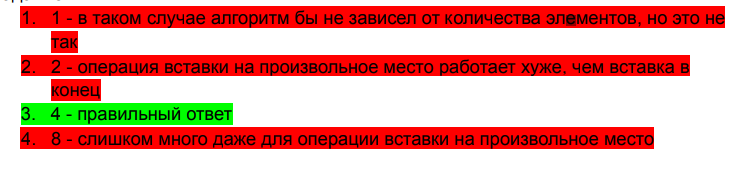
Ну в среднем все же 2.

Сложность такая же. ведь. 2n + константа наверно. Лан вот ответы:

**В среднем увеличилось в 4 раз. Сложность вот:**



**Но я не понял, как получить ответ 4. Вот типо объяснение:**

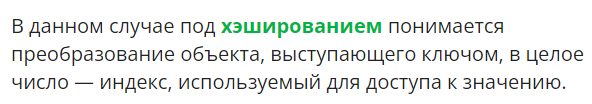


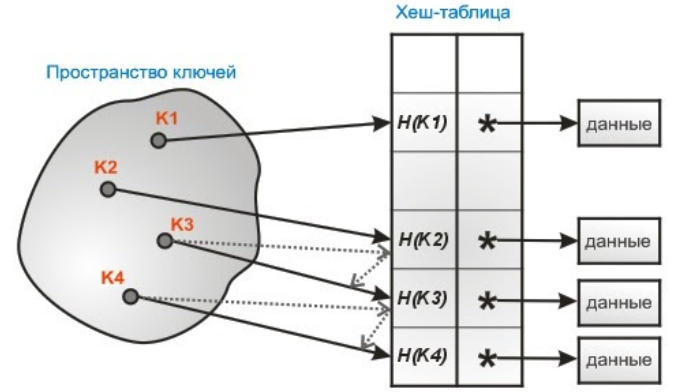
**В остальном познавательно.**

**Хэш-таблицы**

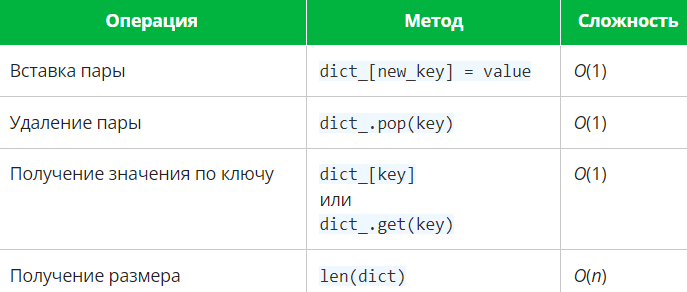
Массив нумерация которой не имеет смысла. Как я уже говорил, объекты, словари и тд.

Словарных тип данных использует такую структуру как Хэш таблицы. Думаю именно так она называется в общем, хотя на отдельных языках может называться по своему.





Из-за условий использования ключ должен быть неизменяемым и уникальным. В таком случае не будет никаких проблем с тем, чтобы получить по нему данных.



Может показаться, что раз уж практически у всего сложность константная, то почему хэш таблицы не используют всегда вместо массивов? Дело в том, что хоть сложность и константна. Но операции более тяжеловесны как по памяти, так и по времени, нежели другие структуры данных.

Тем не менее они очень удобны, особенно удобны при работе с базами данных. Надеюсь доживу до них.

**B9.4. Основные структуры данных: список, стек, очередь**

Стек, да помню. Всегда сравнивал его с патронами в магазине. Так, перерыв на мангу. Не видимо сегодня все.

## ****Списки: односвязные и двусвязные****

Ну.. Это типо матриц, или речь о другом?

Кое что интересное о списках. **Список может быть хаотично распределен в памяти.**

Порядок в этой структуре данных задается наличием указателей на следующий и/или предыдущий элемент в списке.

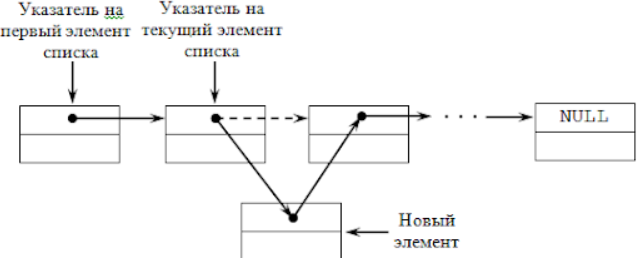


Если в каждой ячейке памяти хранится указатель только на следующий элемент, то список называется **односвязным**. **Хорошо, допустим. Но пока не уверен. Разве списки в python не связаны указателями с двух сторон, ну типо -1 элемент списка это последний. Пока не уверен.**

Если указатель и на предыдущий и на следующий, то это **двусвязный список.**

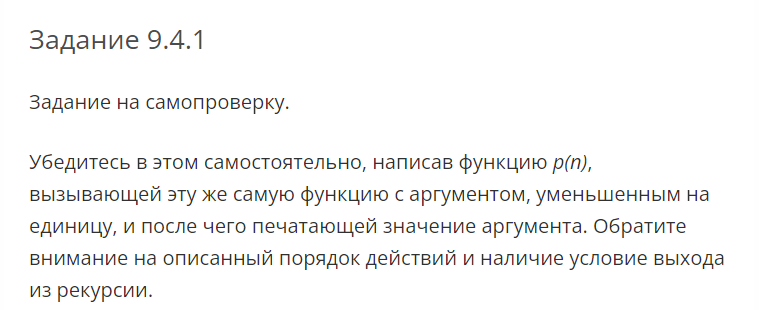
В ячейке также может хранится индекс – порядковый номер элемента в списке. **Но доступ к элементу списка по индексу, сильно отличается от того же в массиве, из-за особенности хранения. Тут полагаю речь о том, что у списка данные могут быть распределены по памяти в разных местах, пока догадки.**

**Плюсы и минусы способов хранения у списков.**

1. Вставка в конец списка происходит за константное время, если в первой ячейке хранится указатель на последний элемент.
2. Со вставкой в начало схоже, но объяснение не понимаю: **“ведь достаточно в новом элементе вставить указатель на тот, что был первым, и дописать указатель на последний элемент.”**
3. Вставка в произвольное место, что-то вроде этой картинки. 
4. **Удаление из начала тоже константное.**
5. **А вот удаление из произвольного места линейное. Так как искомый элемент придется найти по индексу или значению, а значит по списку придется пробегать. Даже если удалять последний, все равно придется пробегать весь список, так как в первом элементе нужно обновить указатель на последний элемент. Но тут речь об односвязазных списках. У двусвязных списков такой проблемы нет.**
6. **С таким способом хранения не требуется переносить всю структуру при расширении списка.**
7. **Общий размер списка может храниться в самой структуре. В таком случае это значение можно получить за константное время.**

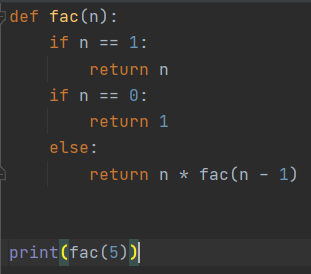
## ****Стек (stack)****

Принцип схож, упорядоченный набор элементов. Как список или массив. Но есть только 2 операции. Положить сверху или взять сверху. Из примера, рекурсивная функция вычисления факториала. Будет вызывать саму себя, но с меньшим факториалом, до тех пор, пока этот факториал не будет равен одному.



Ну коли сам сказал лады. Напишу функцию вычисления факториала, в который уже раз…

Да.. детская задача. Но все равно минут 7 потупил.

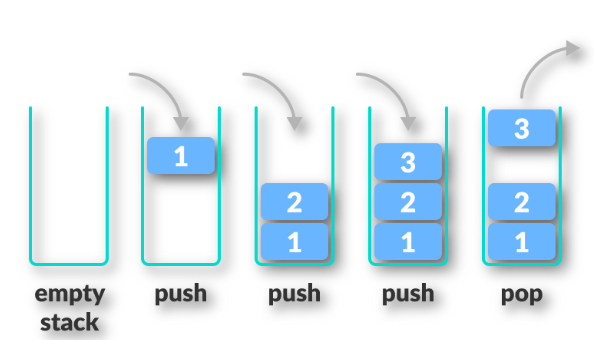
Напомню, что это стек типа **LIFO.**

**Last in first Out. –** Последний вошел, первый вышел. Далеко не отходя, да. Есть еще **FILO.**

**First in last out. –** Первый зашел, последний вышел. Стоп… Это же тоже самое. Похоже я перепутал. Он наверное называется

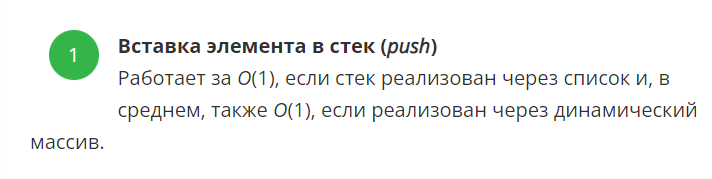
**FIFO – first in first out.** Ну то есть обычная очередь. Но это я из памяти. Продолжу.

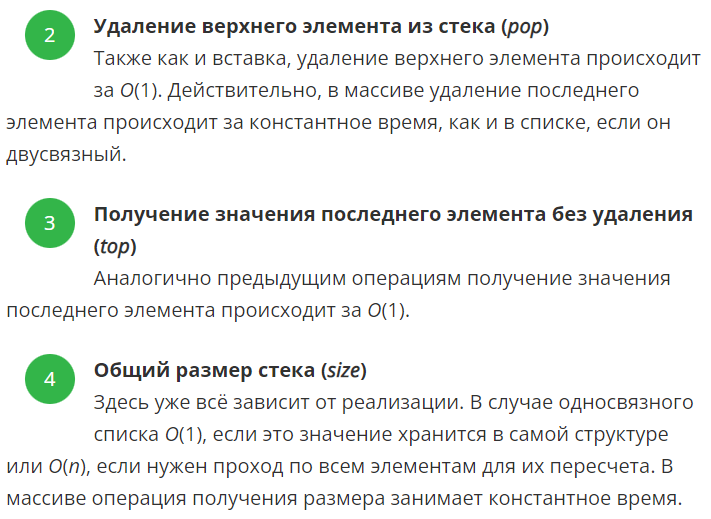
Вот типичная картинка стека:



## ****Операции над стеком****

Реализован может быть как через динамические массивы так и через списки. Но сначала о его механизмах.



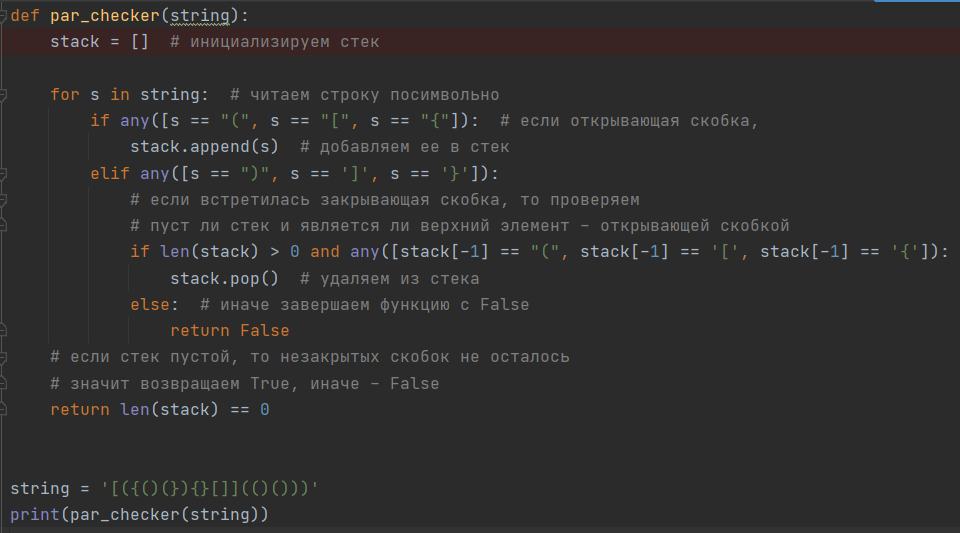


Понятно.. Решил не переписывать так как информация доступна.

Так тут функция проверки скобок в строки. Чтобы они все закрывались через стэк. Не до конца ее понял. Посмотрю.

Понятно. Работает так. Когда видит открывающуюся скобку, кладет ее в стек. А когда видит закрывающуюся проверяет не пустой ли стек и является ли последний элемент стека

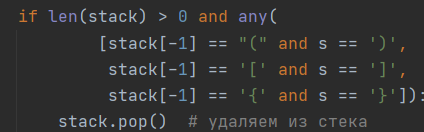
Вот модифицированная версия этой функции, которая учитывает, все виды скобок для строки.





Вот черт. Моя прогу просто идет по количеству. А тут можно устроить учет корректности кода.

А в моем ошибка из-за нижнего any. Там нельзя выбирать все. Должен быть конкретный. И связанный с текущим типом скобки. Без словарей наверно можно исправить так:?





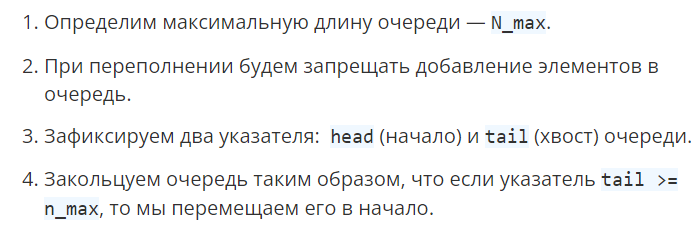
Это решение превосходно.

## ****Очередь****

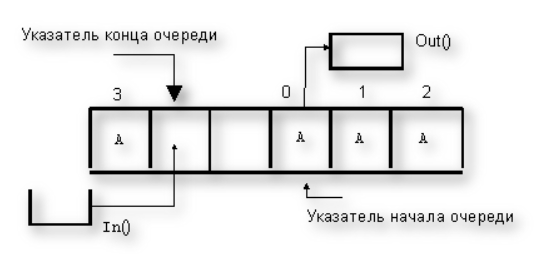
Пока на этом остановимся. И продолжим. Обычная очередь. Или как я уже писал **FIFO – first in first out.** Ну что сказать электронная очередь, очередь в магазине.

Очередь имеет свое ограничение, так как удаление из конца или вставка в начало имеют сложность O(n)

Но кажется эти ограничения можно обойти. Вот как:



Последнее пока не до конца понятно.



С картинкой пока тоже не понятно. 1, 2. Полагаю пока не в очереди. Нулевой элемент это последний парень в очереди. В таком случае Если известна длина, в данном случае 3, да даже без нее, нужно просто убирать последний элемент. А все остальные двигать вперед. Ладно не буду угадывать.

Вот операции в очереди:

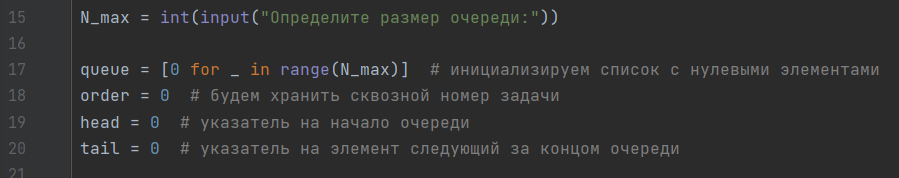




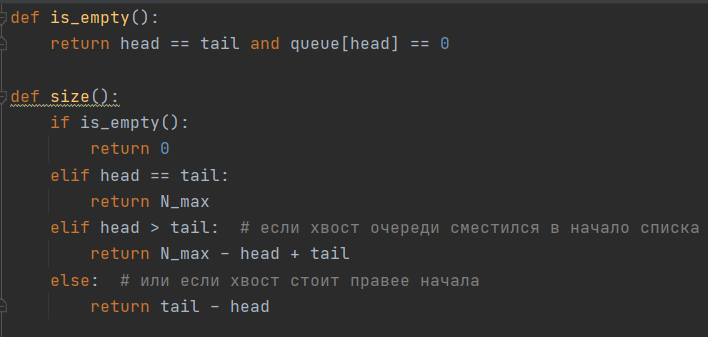
А теперь разберу данный мне пример.

Блин, не до конца понимаю ее концепцию. Без пробега в отладчике трудно будет.

Для начала вот такие данные:



Вот пару функций.



**is\_empty почти полностью понятна. Второе условие как я понял нужно для полностью заполненой очереди. Потому что там, head будет равен tail.**

**size первые 2 условия теперь тоже понятны. Но вот остальные… Сейчас.**

длина 4.

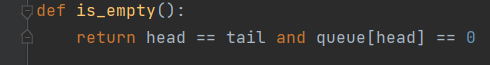
head == tail == 0?

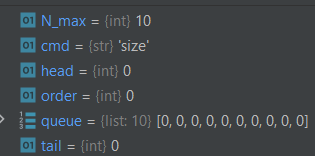
[0, 2, 1, 0] – а это допустим указатели?

[0, 1, 2, 3] – допустим список

Ладно, продолжу. Пока не понятно.

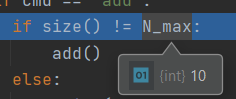
Разбираем по полкам. Функция is\_empty при пустом очереди длиной 10.

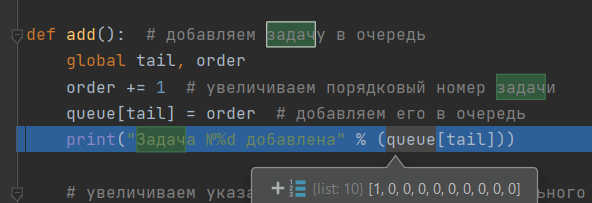




Посмотрим как работает **add.**

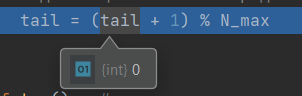
**Но сначала идет проверка. До size еще не дошли, но он возращает текущий размер очереди если правильно понял. А тут проверяется не забита ли очередь до конца и если нет. 0 != 10**





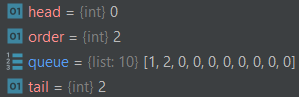
Сначала **order** увеличивается на единицу. **tail** – хвост не изменялся и указывал на нулевой элемент. Первый в этом списке. Туда записалась единица как первая задача.

А затем изменяется указатель на хвост. Был ноль, а теперь будет на 1 больше.



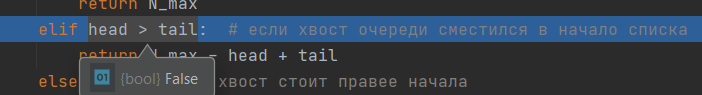
Причер когда он достигнет N\_max, ну то есть 10. Он снова будет равен 0. И так по кругу.

Вот ситуация после 2-х add.



Попробую зайти в size.

Мне интересно когда это условие выполнится.

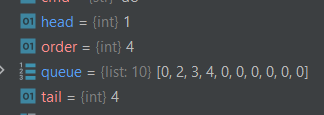


Пока просто последнее:



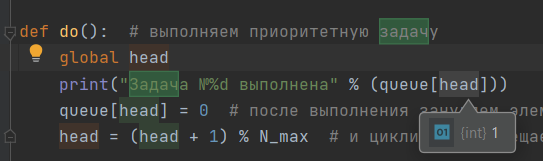
2 – 0 = 2

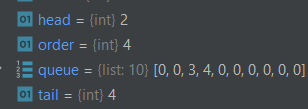
Добавлю еще 2 задачи и одну выполню.



head изменился. Если сделать size то будет 4 – 1 = 3. И в очереди видно 3 задачи. Теперь внимательнее рассмотреть еще один add.

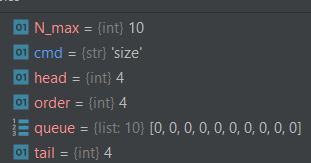
do выполняет задачу текущего head’a, а затем увеличивает его. И точно также если доходит до максимума, в данному случае до 10, то снова приравнивается к нулю.





А теперь добавлю 5 элементов.

Я дурак. Вместо добавления 5. Я выполнил все.



Вот оно. Спец условие когда head > tail.



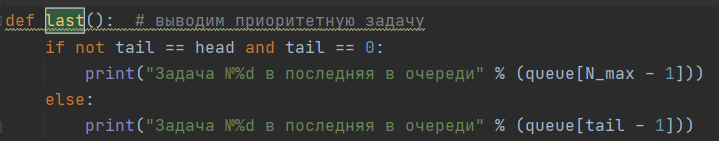
Ничего никуда не придется сдвигать. Оно просто запишется в свой хвост.

Вот терь Size:



10 – 5 + 1 = 6

Так как я эти функции только разбирал. А сам не особо писал, то я тебе придумал задачку. Этими же средствами получать последнюю задачу в очереде. Вот эта функция.

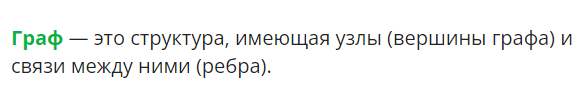


Это хорошие знания.

**B9.5. Нелинейные структуры данных: графы и деревья**

## ****Графы****

Ух, дискретная математика.



И сразу еще базы данных вспоминаются, что очень хорошо.

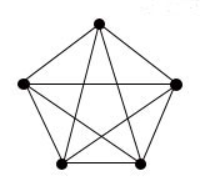
Графы бывают разные. В этой статье подробно

<https://function-x.ru/graphs2_definitions_classes.html>

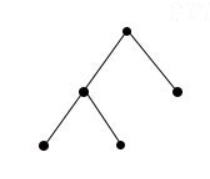
Сейчас ее прочту.

Просто пару примеров.

**Полный граф. Все вершнины соеденины медлу собой.**



**Граф-дерево.**

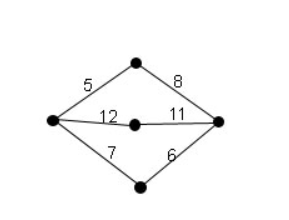


Число q рёбер графа находится из соотношения

q = n - 1,

где n - число вершин дерева.

**Взвешенный граф. У ребер есть свои веса. Хм.. мб и я дойду до нейросетей.**



**Еще речь шла о ориентированных и неоринтерованных графах. Не совсем понял, но у ориентированных есть дуги и важен порядок концов этих ребер. У неоринтированных наоборот.**

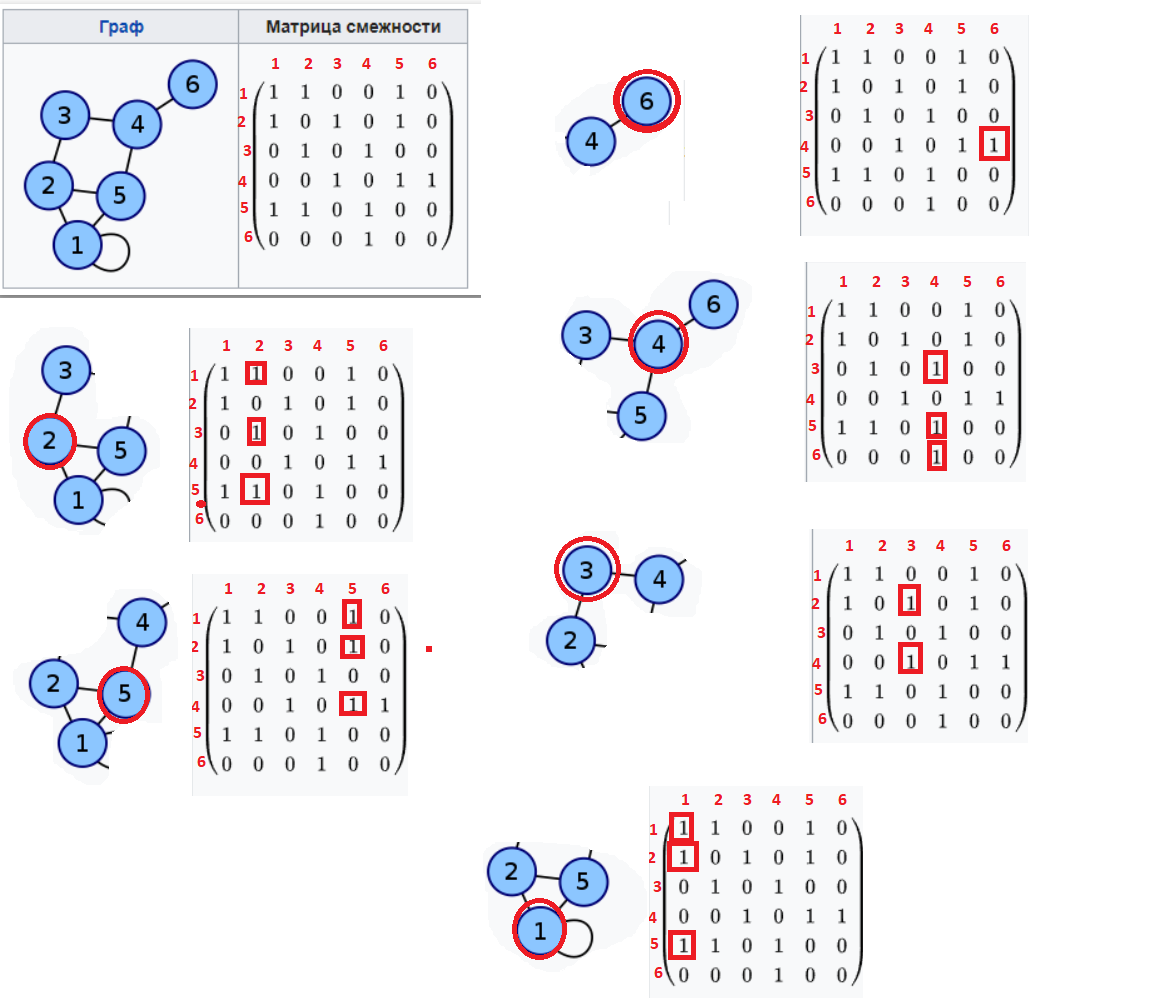
Продолжим. Как же их представить на компьютере. Способ 1. Матрица смежности.



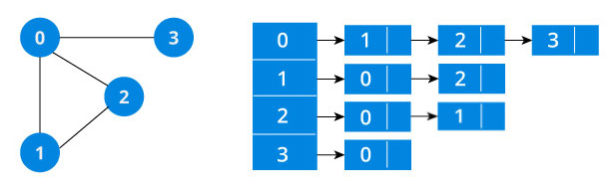
Попробую подумать. Как минимум это матрица 6 на 6. И у нас 6 вершин.

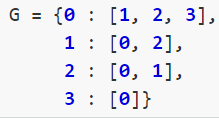
Порядок конечно странный, я бы начал отчет с той позиции, который сейчас 6. А закончил бы позиции 5 или 3, одна из них был 6-ой.

Так вот. Матрицу я разобрал самостоятельно. Достаточно элементарная. Я шел по столбцам. Но разницы нет, на то она и матрица.

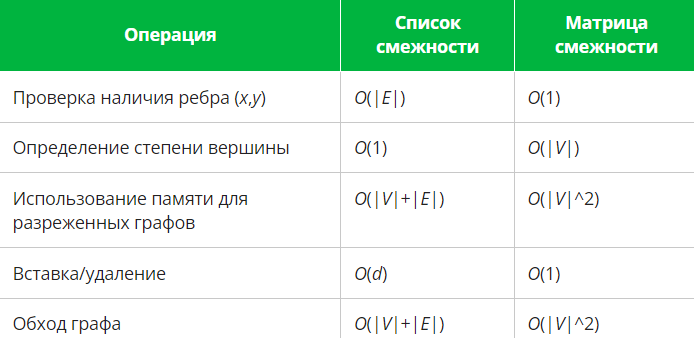


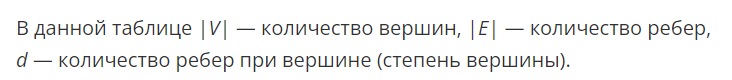
Во пример по проще если необходимо. Тут даже со словарем заполненным.



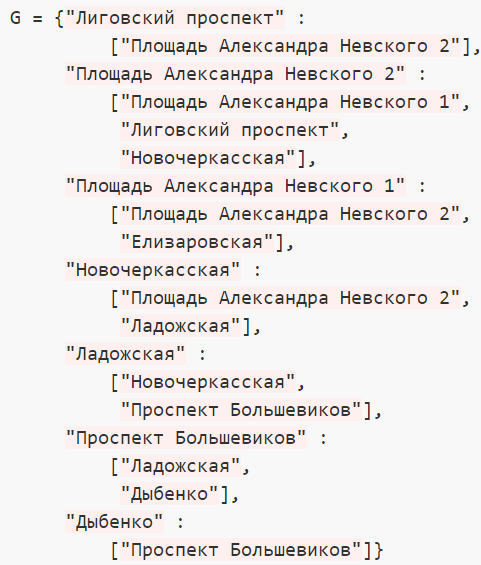


По сравнению с двумерными массивами имеют невысокую сложность.



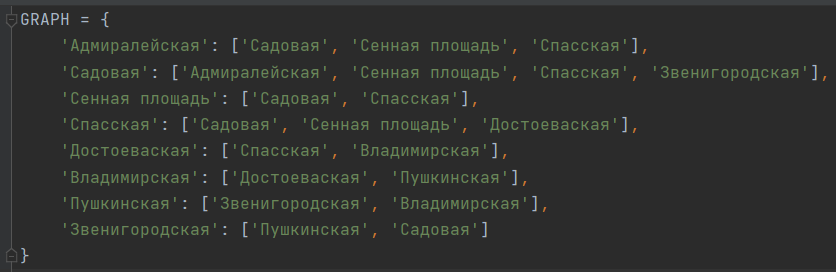


Вот так рисунок метро можно представить в форме такого графа.

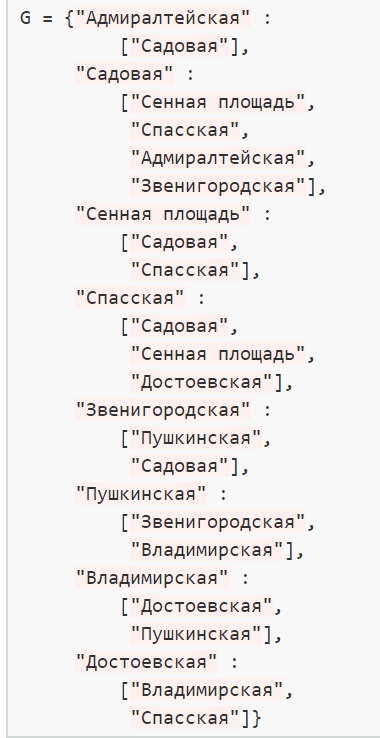


Лааадно. Сделаю.

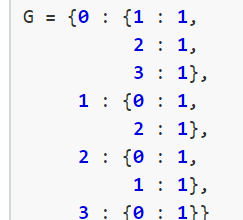


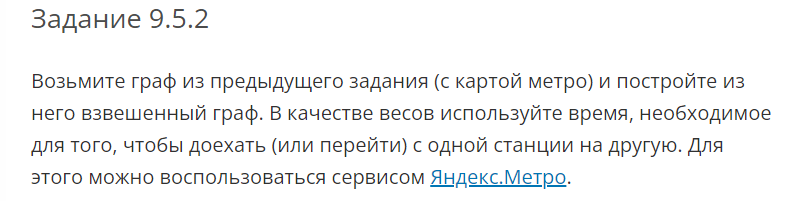
Не самый маленький. А это всего лишь маленькая развязка. Лааадно~

Допустил ли я ошибки? Да! Но всего одну. Адмиралтейская фиолетовая, поэтому связана только с садовой. Протупил.

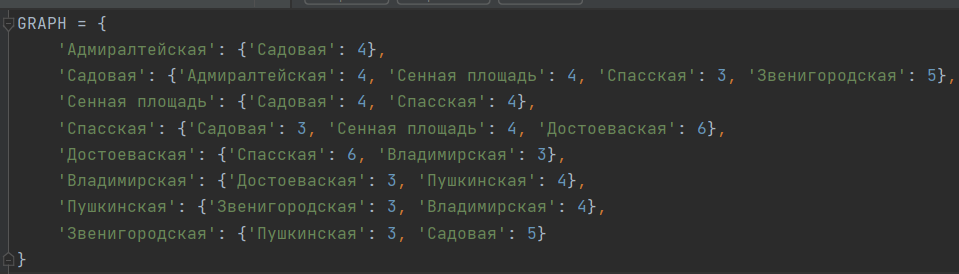


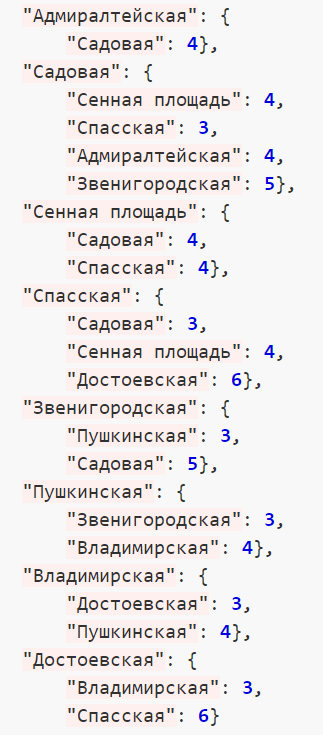
Чтобы представить взвешенный граф, то надо просто внутри ключа вершин хранить словарь, в котором ключи это связанные ячейки, а значения это веса. Пример:





Хуя вы… ну ладно попробую.



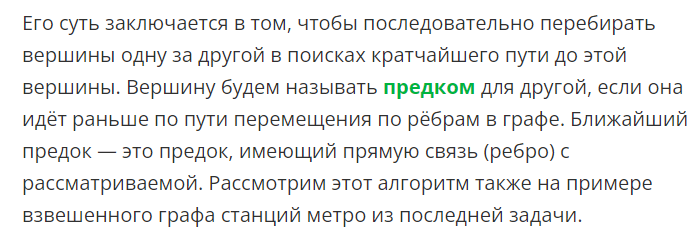


Ну вроде все верно. Блин устал чутка.

Наиболее популярная задача с графами это нахождение самого быстрого пути. Перерыв.

Для этой задачи существует следующий алгоритм.

**алгоритм Дейкстры**



Для начала из графа получим все ключи вершин. И зададим им все значения. 100. Просто потому что оно заведомо больше любого веса.



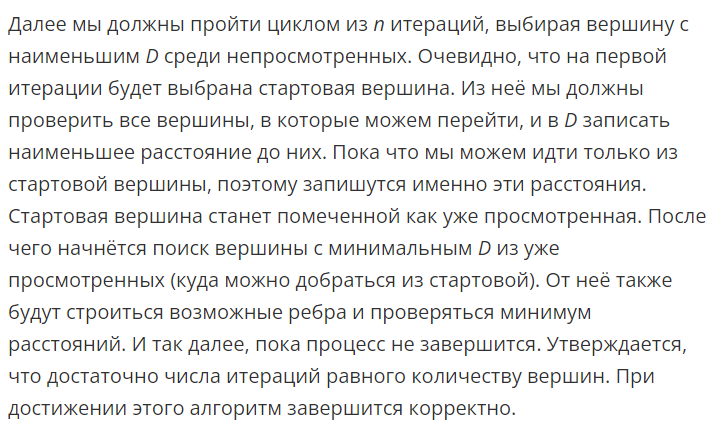


В начале нужно выбрать стартовую вершину и ей задать вес ноль. Так как растояние от нее, до нее самой ноль, наверно?

И еще словарь для хранения булевых переменных в этих вершинах.





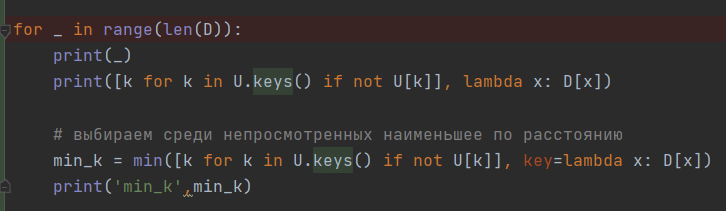


Полотно. Вообще не воспринимается.



Блять вы понимаете как сложно воспринимать такие строки?

Даже отдельно расписав. Если вы нормально ее не объясните иди нахуй, потому что вот я вывожу для упрощения:

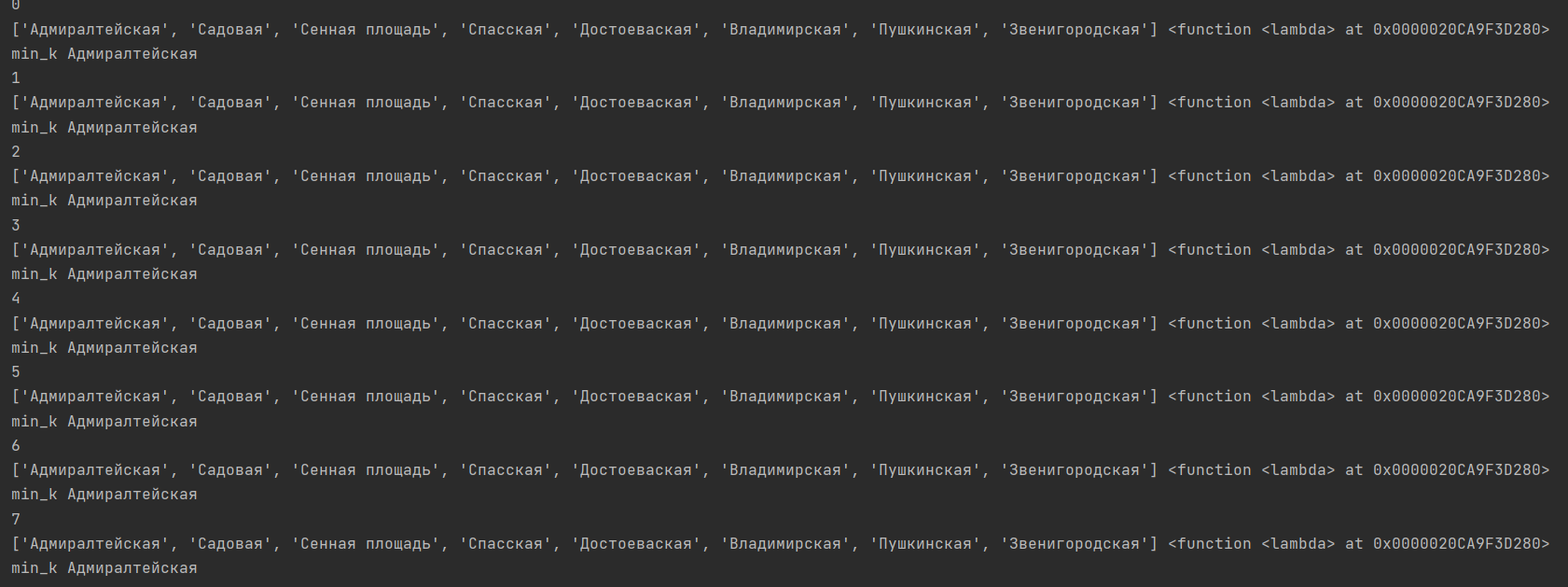


\_ = 0…7

Сначала просто создается список ключей. Если у станции FALSE. lambda x: хуй пойми зачем.

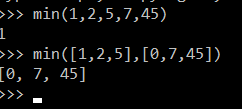
Ну и вишенка за все 7 раз min\_k так и остатется Адмиралтейской.

Что за бесполезная пробежка?



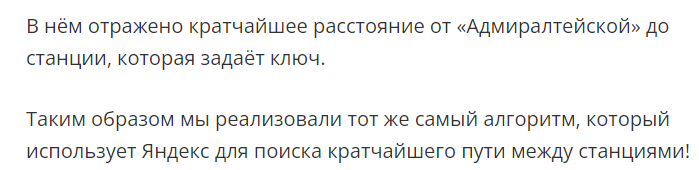
Ааа.. внутри еще одну пробежку нужно сделать. Бээ. z

Ладно по чуть-чуть. Во первых напомню. min() может принимать аргументы сравнения через запятую.



Сравнивает и списки и отдельные цифры.

Кстати мельком глянул объяснение кода



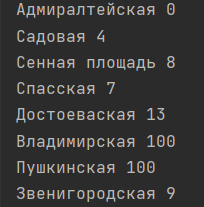
Ой простите, я хотел сказать отсутвие объяснения. Разберу позже, время курить калик. Муахаха.

Хаха.. и снова понедельник. Насколько я помню я разбирал программу по строчно, но не закончил.

Во первых выше я писал про лямбда функцию которую не понимал да?



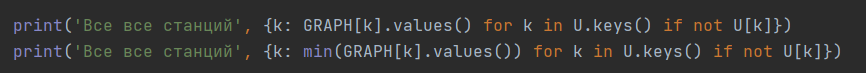
Так вот. Если ее убрать финальный вывод будет таким: а с лямбдой таким:



Она нужна в этой строки:

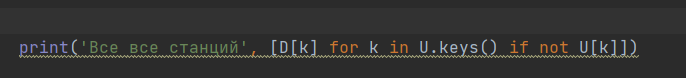


Для критерия min. Чтобы из этого списка выбирался min не по какому-то критерию по умолчанию, не знаю, может для ключей это длина ключа или типо того, а именно по размеру веса. То есть это формально должен быть список весов. Я бы даже сказал эту строку можно написать без лямбды функции как-то так.



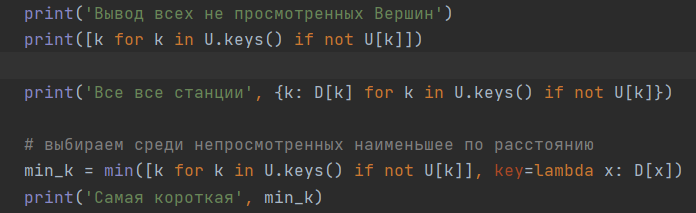


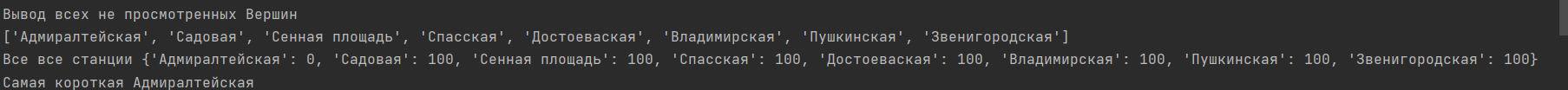
Хотя не, на самом деле вот так:





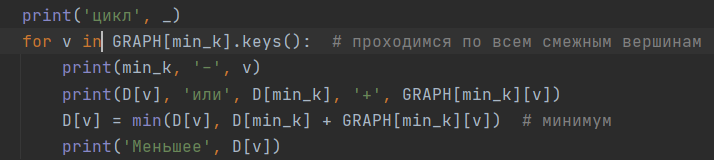
И из этого списка выбирается наименьший. Так еще раз покажу код:



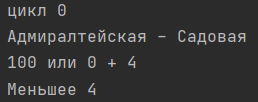


Да, я все это время разбирал одну маленькую строчку. Для min\_k. Он выбирает самый короткий. А в начале программы у всех кроме стартовой позиции будет ноль. Поэтому в начале выбирается она.

Затем пробегаемся по всем граням связанным с этим путем.



А Адмиралтейской к примеру это только Садовая.



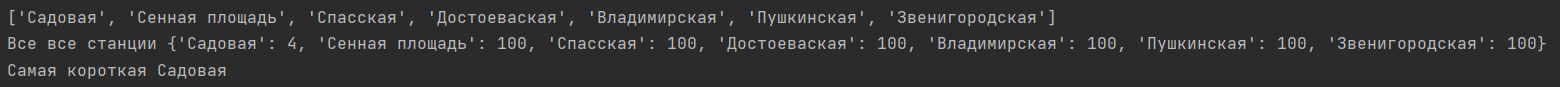
Изначальная 100 или сумма весов обеих. Сумма оказалась меньше.

Адмиралтейскую помечаем как просмотренную.

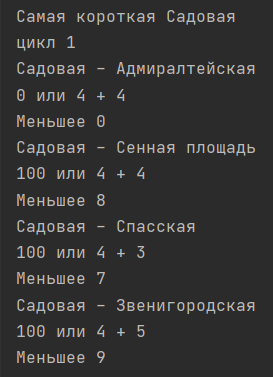




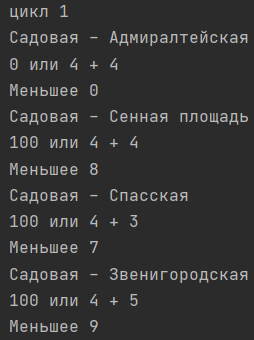
Поэтому в следующем цикле ее уже нет:



Теперь пробежка от садовой.

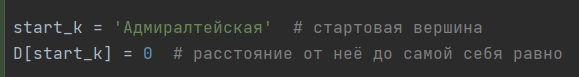
 И что же происходит после этой записи в весах? Выведу их после каждого цикла.





Здесь Садовая – Адмиралтейская ноль, потому что она напомню, стартовая точка И D[start\_k] = 0

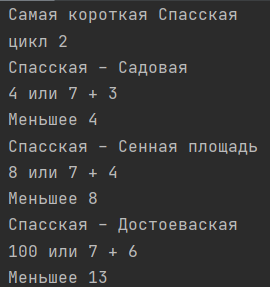




А тут мы как раз именно через D[] сравниваем.

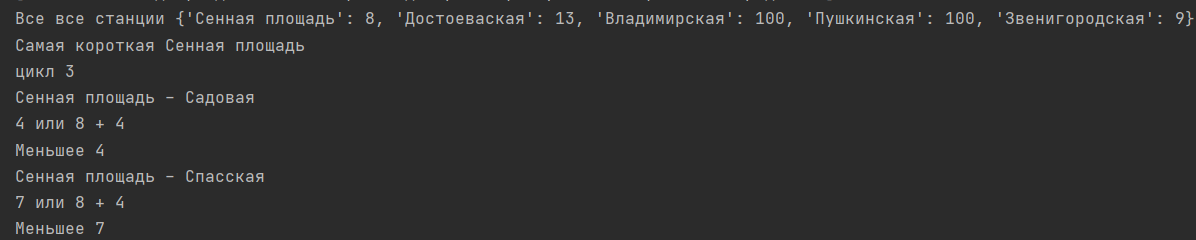








Тут напомню, что выше на скрине это все пути от Спасская. Но в следующем цикле будет выбран не просто наименьший, а тут, который не выбирался до этого. Садовый был, поэтому Сенная площать.



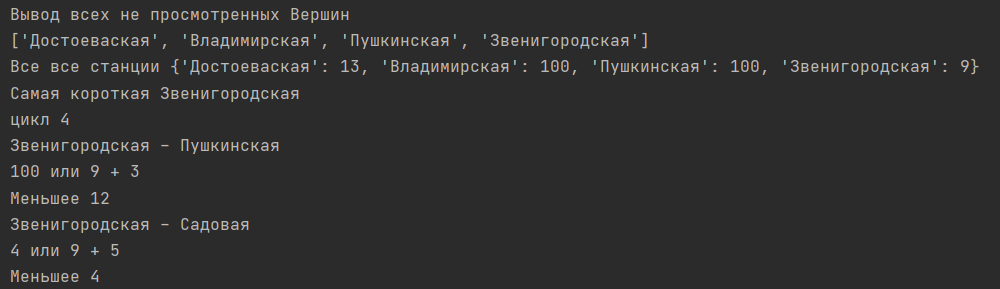
Ошибка у меня кажется. Так как снизу. У меня Сенная – Спасская не 7, а 4.

Потому что не так вывожу. Надо вот так:

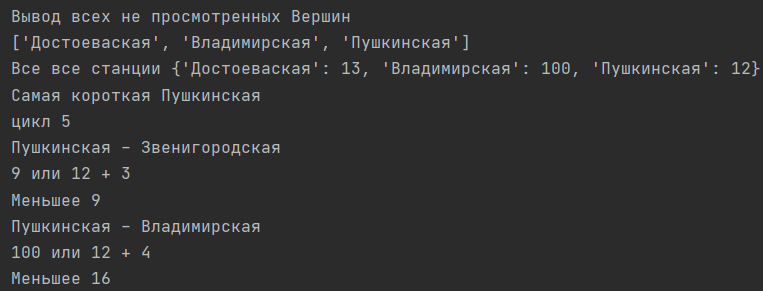


Причем уже после проставления весов. Я заменю картинки выше с подписью: Все вершины пути от… но до этого они выглядили как-то так:

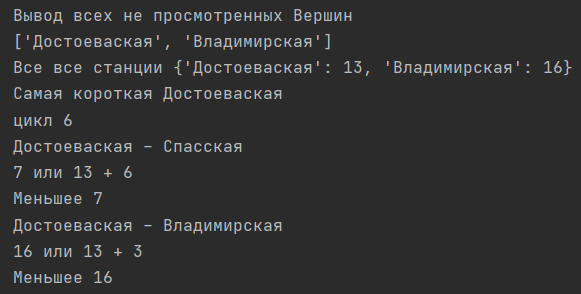




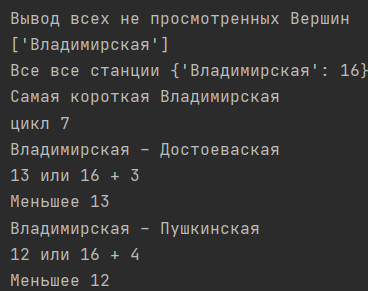


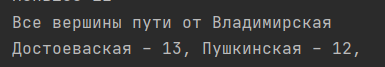




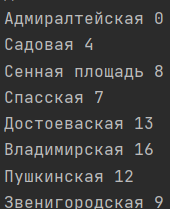




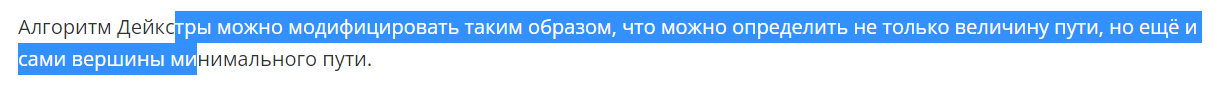




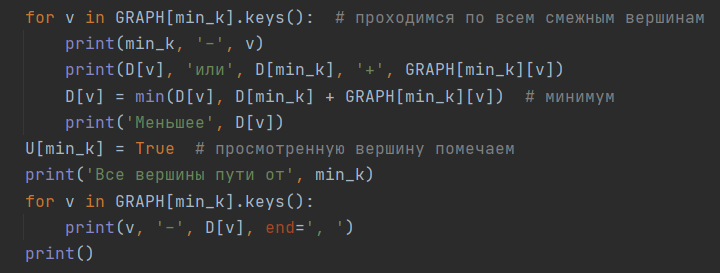
Вот прошли все циклы. И теперь можно вывести расстояния D. Который в финальном состоянии хранит растояния от Адмиралтейской до все остальных вершин, причем самые которкие пути. Правда хранит именно растояния. А не сам путь.

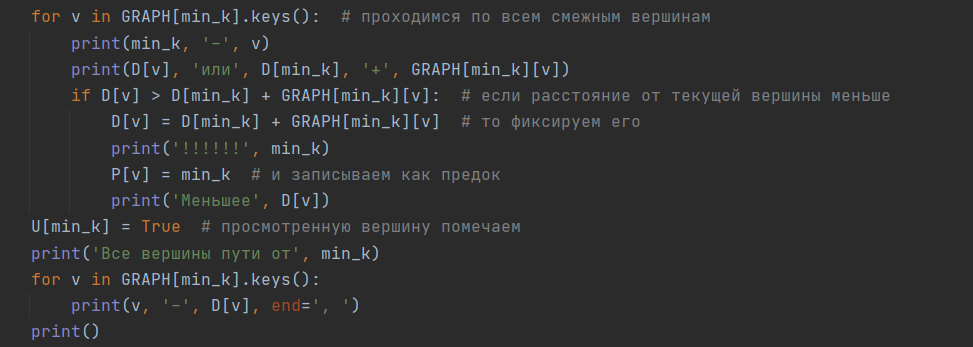


Фух. Что-то было сложно но суть я вроде уловил.

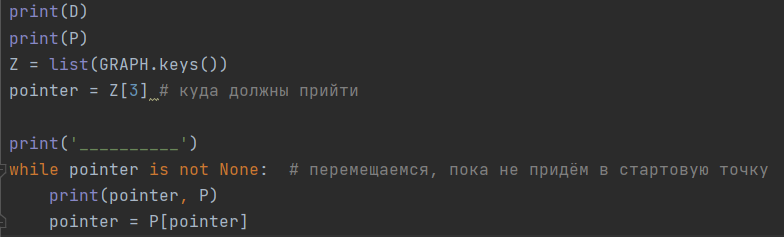


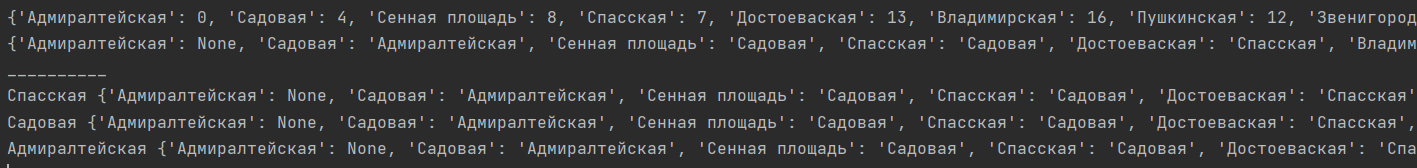
О.. это любопытно. К сожалению я туплю, алгоритм сложный. Поэтому я просто скопировал. Вот оба варианта для сравнения:





А вывод здесь.





По P – видно из какой точки начинаем. Адмиралтейская, а значит в нее будем стримится. То есть в цикле while будут перебраны все комбинации пока не прибудем в точку Адмиралтейкая. От точка Z[3] – Спасская.

P[Спаская] ведет в Садовую.

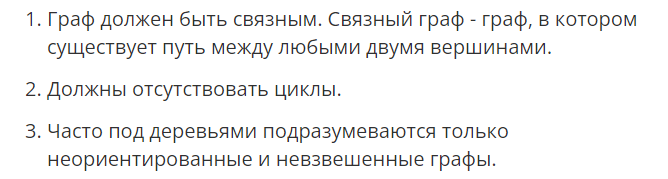
А Садовая в Адмиралтейскую.

А Адмиралтейская – Адмиралтейкая None. Так и работает.

Надо понимать этот алгоритм лучше. Обязательно вернусь к нему если понадобится.

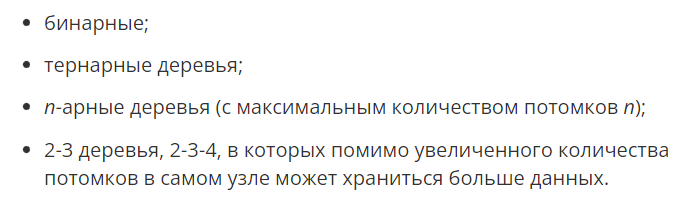
## ****Деревья****

Очень распространенный подвид графов, для его использования сущиствуют несколько важных условий.



Отсутвовать циклы? Интересно. А так действительно похоже на обычное дерево, правда растущее вниз. У него имеются: **корневой узел, потомки, лист(узел не имеющий потомков), внутриние узлы(не корневые узлы, имеющие хотя бы одного потомка).**

В зависимости от количество потомков деревья различают:



Прочту статейку по деревьям. Но конкретно по деревьям 2-3-4.

<https://habr.com/ru/articles/273687/>

Аааа!!! эта статья повела меню на статью по 2-3 кучам..

<https://habr.com/ru/articles/246105/>

В компьютерных науках приоритетом являются структуры в виде кучи. И что это?

Это такая структура данных типу дерева. И она удовлетворяет условию:



Родитель больше или равен потомку. Звучит логично. У кого наибольший ключ, тот и корневой узел.

Кучи имеют решающее значение в некоторых эффективных алгоритмах на графах, таких как алгоритм **Дейкстры** и сортировка методом **пирамиды**.

Хорошо зарекомендовали себя такие структуры как: **Бинарная куча** и **Куча Фибоначчи.**

Но у них свои особенности. У бинарной кучи логарифмическая сложность.



И для слияния таких грапп потребуется линейное время. **Но для хранения требуется мало памяти.**

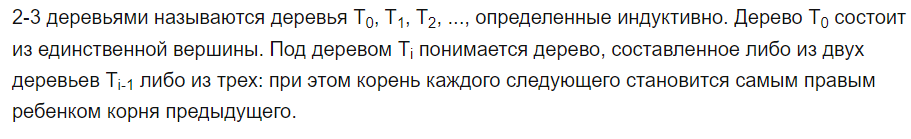
Но зато быстрое нахождение минимального.

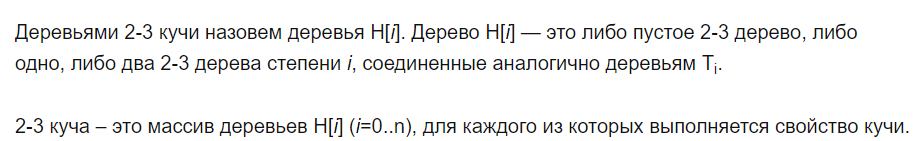
**У кучи Фибоначи** есть балансировка при извлечении узла из кучи. Упрощается сложность основных операций. Но… при большом количестве последовательных операций куча растет в ширину и балансировка происходит лишь при операции удаления минимума. Из-за этого операция получения минимум может занять большие затраты времени.

А тепепь к теме 2-3 куча. Чтобы решить проблему между этими двумя структурами данных, японец Тадао Такаока. (Наверно японец, сужу по имени).

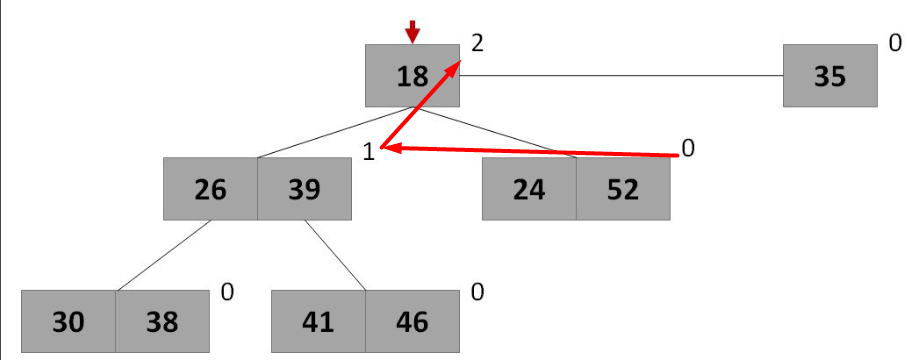
Это альтернатива кучи Фибоначи и состоит из 2-3 деревьев. Суть в том, что балансировка проводится не только при удалении, но и добавлении новых элементов. Тем самым снижается вычислительная нагрузка при извлечении минимального узла из кучи. Количество детей вершины **t** произвольного дерева **T** назовем степенью (**degree**) этой вершины, а степенью дерева

назовем степень его корня.

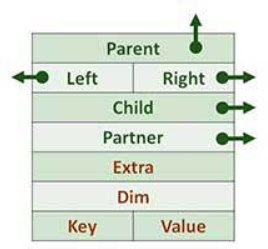




Так тут есть рисунок пример для представлений. Я в нем пометил красными стрелками логическую связь которую я как понимаю, ну или думаю, что понимаю.



Будто от каждой кучи может идти от нуля до 2. То есть 3 кучи максимум.

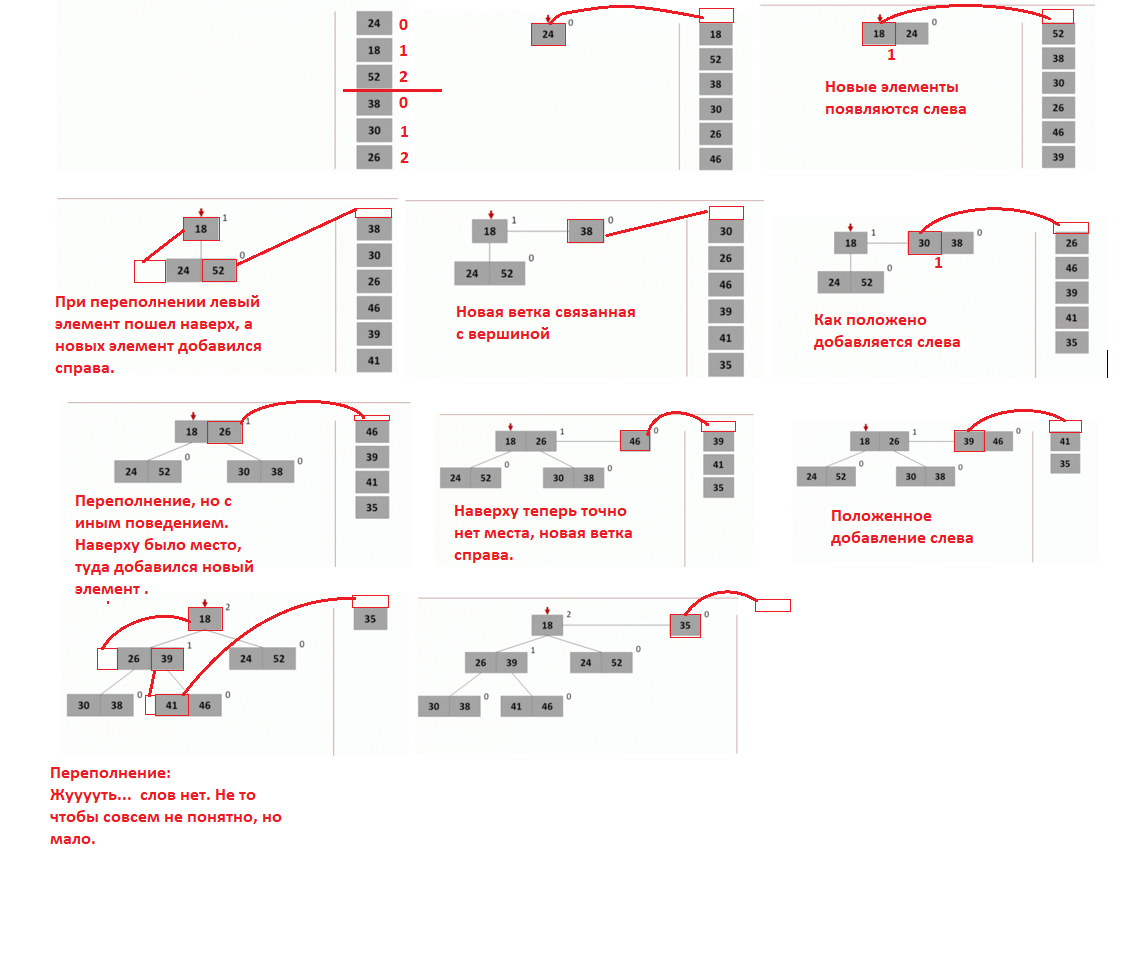




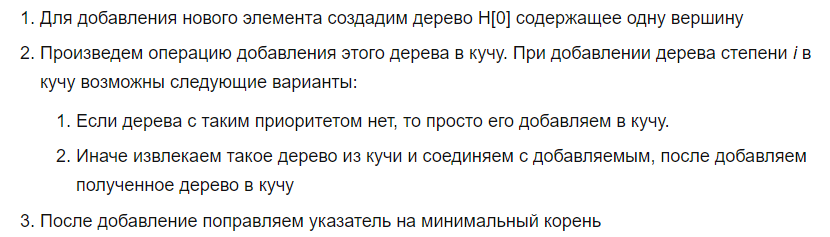
Нахождение минимального элемента кучи проходит за константное время**, Θ(1).**

**Вставка.** Но сначала картинками.

Чтобы самому понять

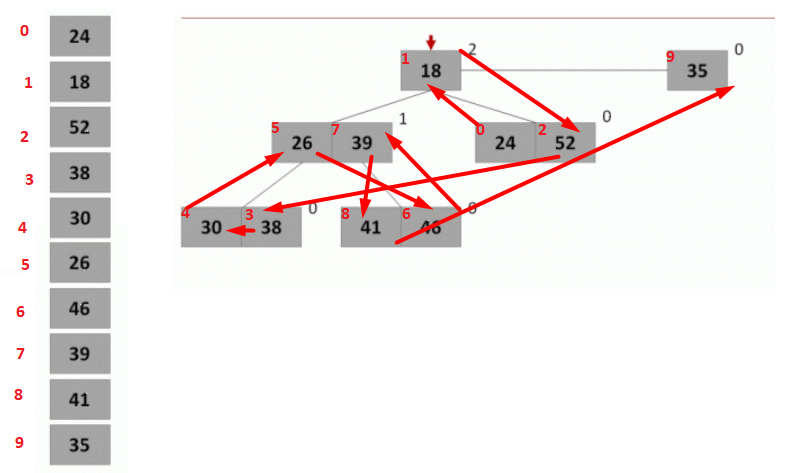


Понять самому? Ахахаха

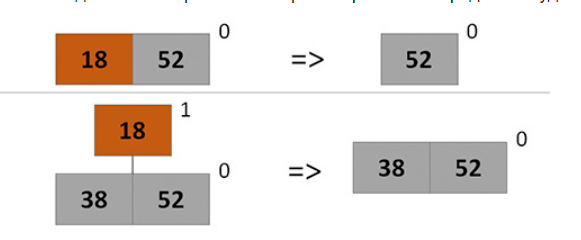


Это не сильно помогает.

ПХахахахаха, тот кто это придумал сумашедший. Я вообще не вижу логики в этой очереди.



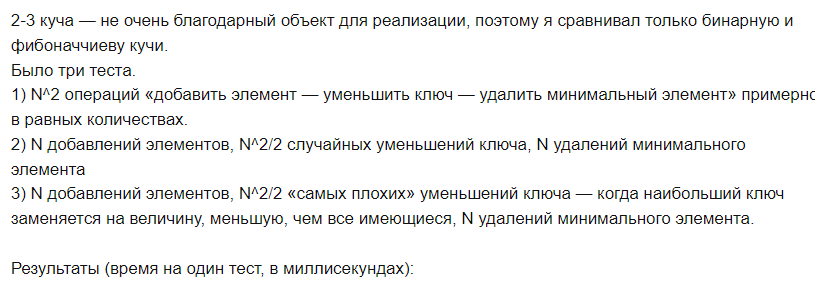
**Удаление манимального, зачем я лезу…**



**Уменьшение ключа у элемента.**

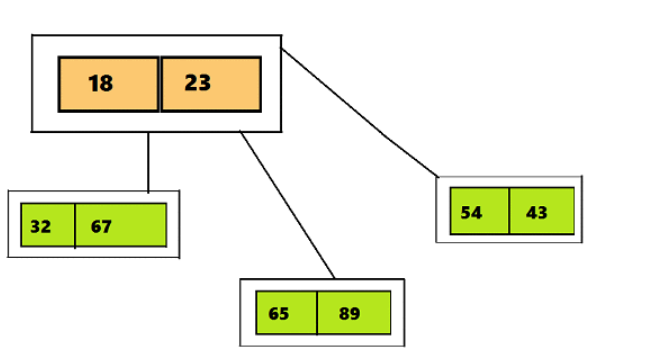
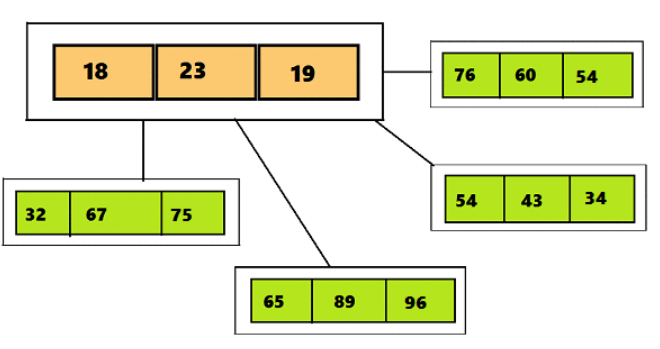


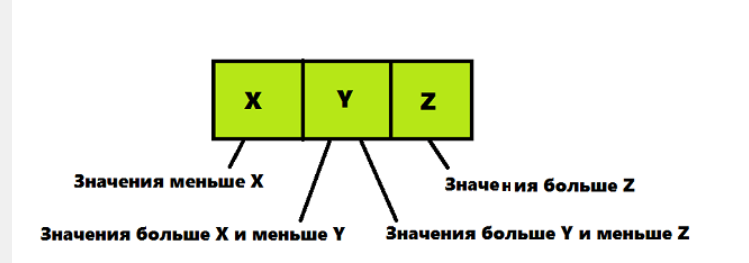
Ладно конец этой статьи 2014 года на хабре. Там в коментах еще тест провели.

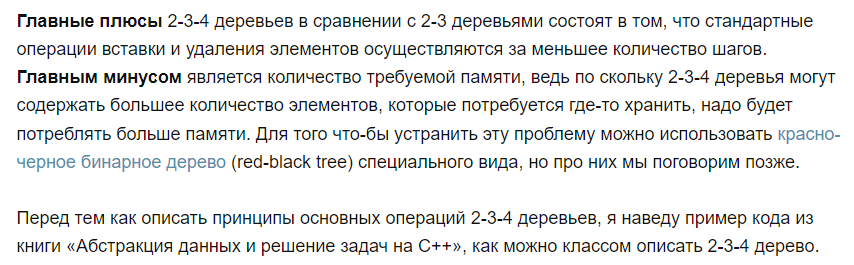
 

А теперь по рекурсии возвращаюсь в статью 2-3-4. Это же еще сложнее может быть. Или может там объяснение для маленьких будет? Пожалуйста..

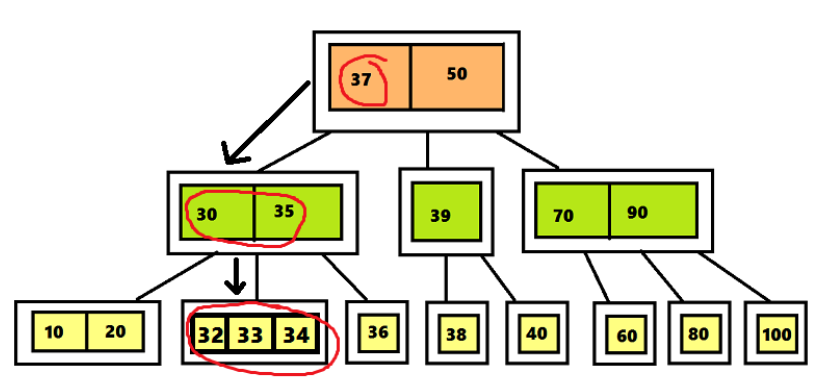
Главное отличие, что дерево может содержать более 3 дохерних узлов.

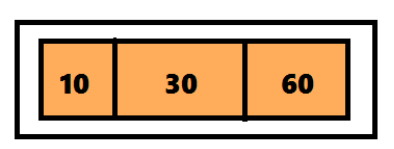
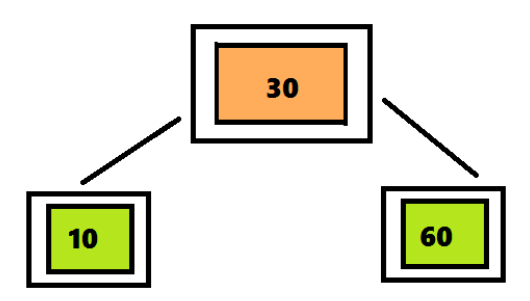
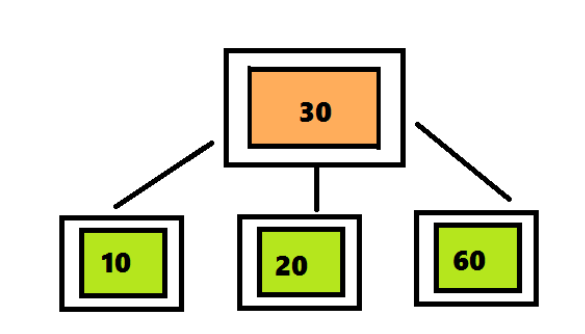
 





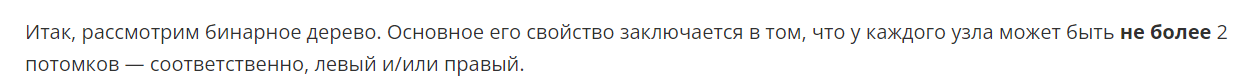
**ВАЖНОЕ ПОНИМАНИЕ. В ЭТИХ КУЧКАХ САМЫЙ ЛЕВЫЙ ЭЛЕМЕНТ НАИМЕНЬШИЙ, А ПРАВЫЙ НАИБОЛЬШИЙ. ВИДИМО ОНИ НУЖНЫ ДЛЯ СОРТИРОВКИ.**



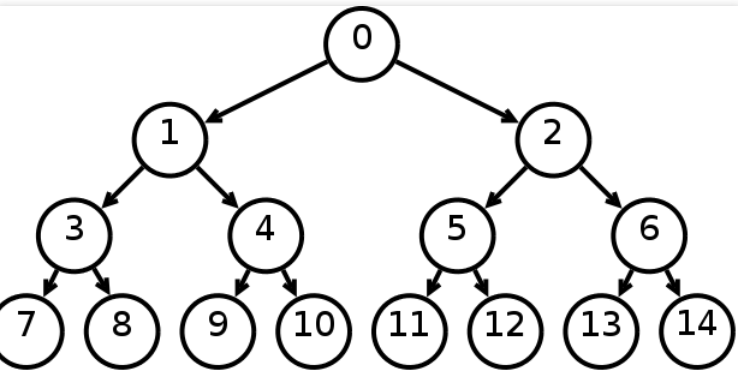
  

Вот бы только нормальный пример кода увидеть.

Лан, продолжим. Трудно изучать обычные 2-3-4 деревья когда даже **бинарное дерево не помнишь или не знаешь. Я не знаю)**



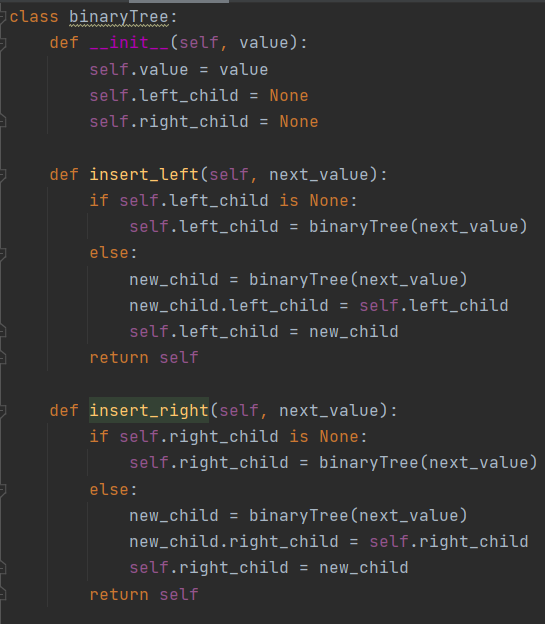
Хм… вот и узнаем что такое бинарный поиск и он ли также пузырьковый метод?



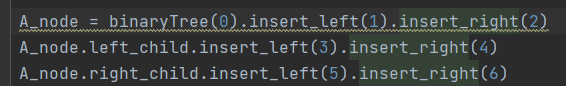
Мега понятная картинка.

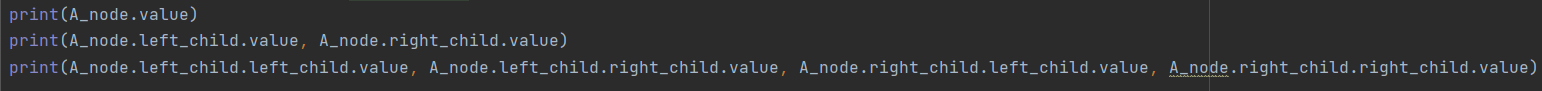


С этим сложно поспорить. А реализовывать эту схему будем с помощью классов. А методику использовать схожую с тем, как в памяти хранятся списки, которые хранят в себе собственное значение и указатель на следующий элемент. Ну я понял, что это будет класс, который хранит свое значение, и потомков, которые будут хранить своих потомков. Попробую что-то набросать.



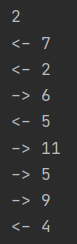
Вот кодик. А вот как с помощью этого класса можно заполнять потомков:



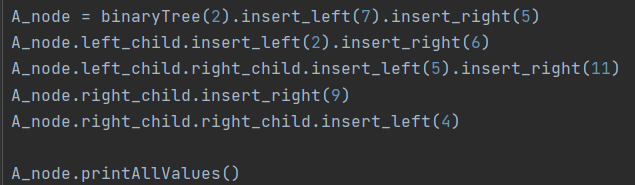




Такс.. а как это в цикл впихнуть?



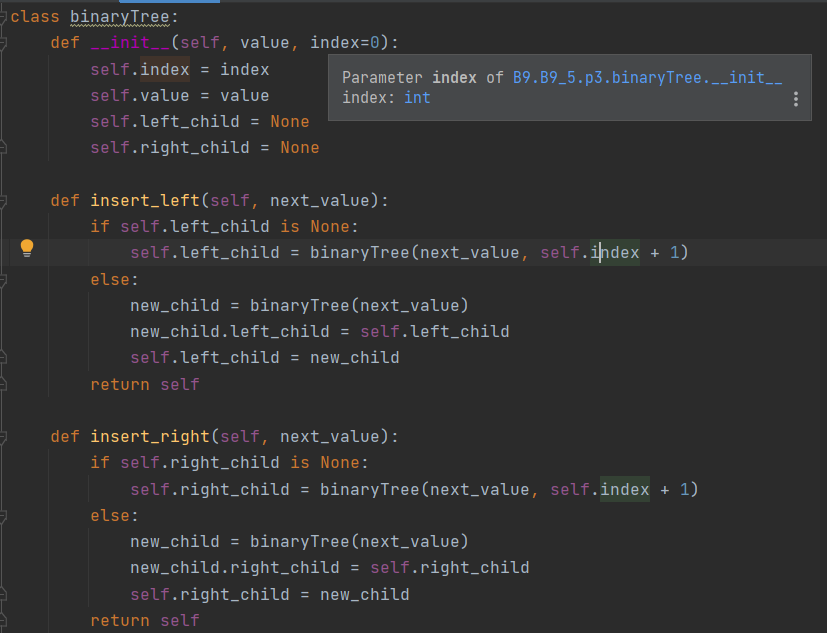
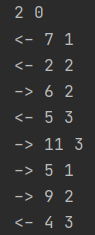
Пока только ручным заполнением…



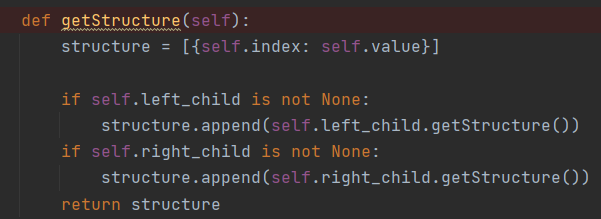
Зато я написал функцию для вывода дерева!



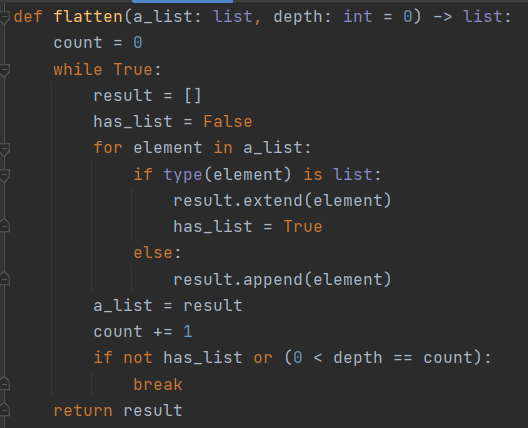
Доработал класс таким образом, чтобы он внутри себя еще хранил уровень, на котором дерево находится.



Затем на основе функции вывода нарисовам функцию возвращающую структуру.



Украл небольшую функцию для распаковки вложенных списков.

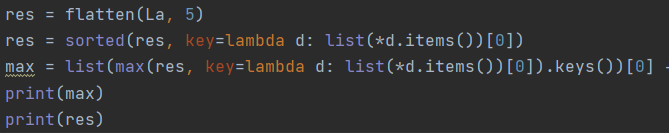


Функция getStructure возвращает дерево в таком формате.



Ключ уровень на дереве, а значение это значение. Убираю сначала все вложенности.

Убираю вложенность. Сортирую(Не обязательно)

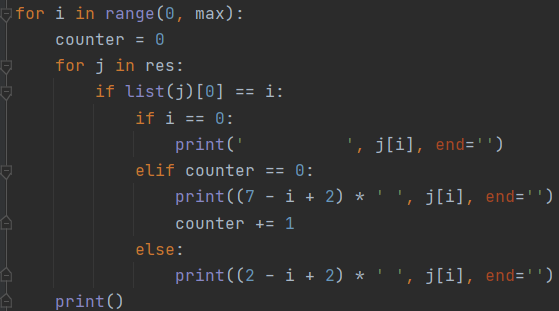


Получу такой список.

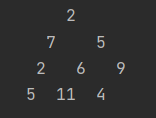


А еще вычисляю максимальный ключ, он нужен чтобы быть предельным для пробежке по дереву.

И пробегаюсь по дереву:



Counter нужен, чтобы понимать это левый или правый потомок. Чтобы дерево рисовалось чуть красивее. Вот результат.



Попробую не много улучшить рисование. Допустим если стартовая точка 10

10 \* ' '

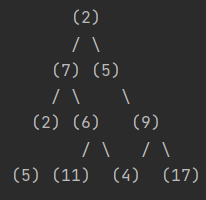
10 – 1 ‘ ’ - 9

10 + 1 \* ‘ ’ 11

2

7

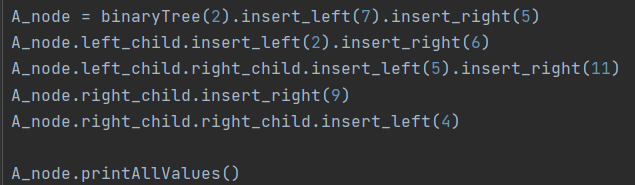
Результат миллион процентов не идеальный, даже я бы сказал плохой. не могу подобрать точные отсупы. Но вот



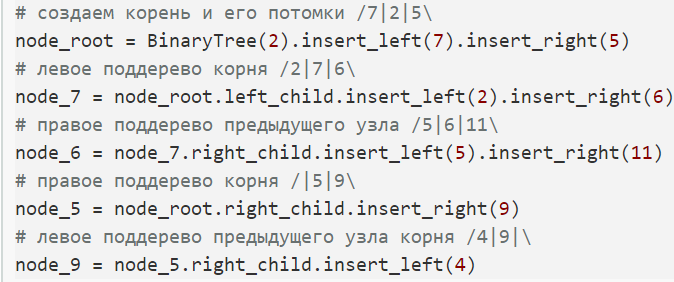
Е мае. Я очень давно не занимался. Даже работал много. Но сейчас нужно отвеличься.

С этим деревом я поигрался.

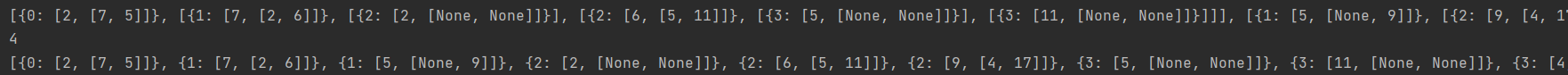
Напомню я дерево это так прописал.



В предложенном ответе каждый узел подписан конкретной переменной. И да, выглядит удобно.



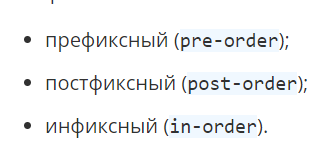
Я уже сделал некоторый вариант обхода такого дерева, но я его дополнительно модернизировал. Сохранял индексы. Писал каким потомком объект является левым или правым и тд.



А какой вариант мне предложат?

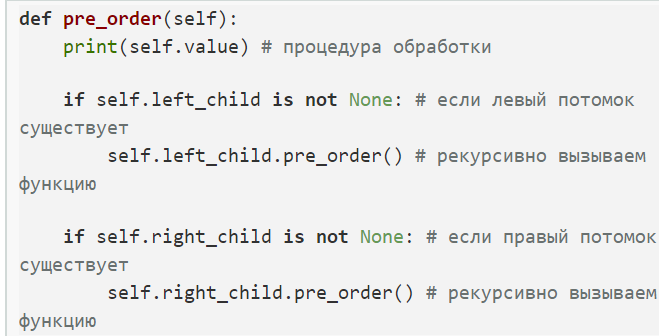
Различают 2 основных подхода обхода деревьев. **Поиск в глубину – DFS.** И **поиск в ширину – BFS**

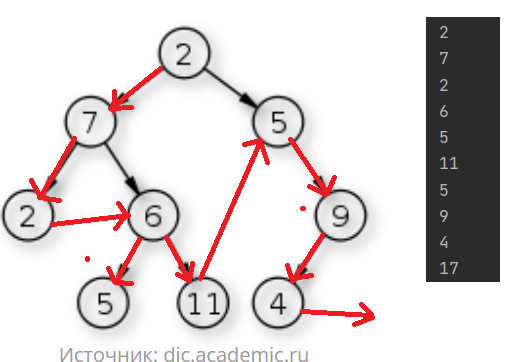
**В глубину** проходят по каждому узлу, а потом возращаются обратно. Обход с возвратом. Бывает 3 видов.



**Поиск в ширину** осуществляется в обходе уровня за уровнем. Разницы по описанию пока не понял.

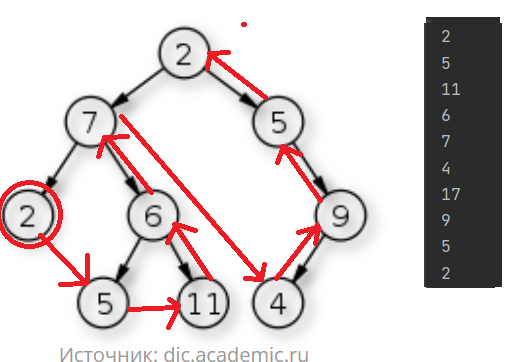
Тут ходит сначала по всем левым потомком до самого края. Затем правые потомки:



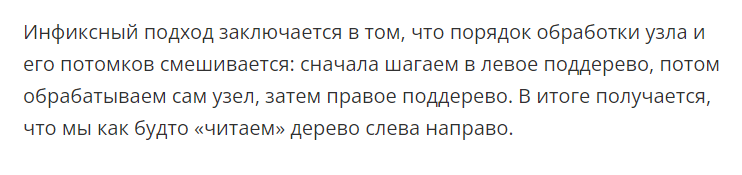
****

**Постфиксный отличается только моментом когда вызывается print. Не в начале, а в конце.**

Запись начинается от первого узла у которого нет потомков.

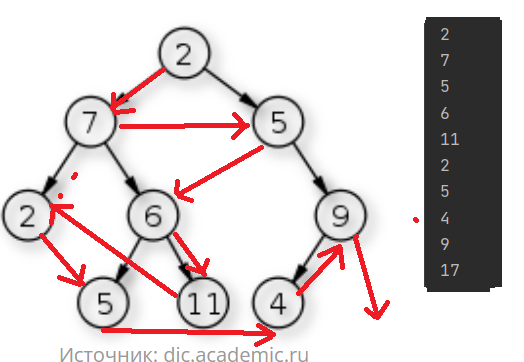


Инфиксный наверно если print писать внутри if.



print между:

Эм… ну и странный вывод.



Ссылка на почитать. Пока завершу.

<https://medium.com/nuances-of-programming/%D0%B2%D1%8B%D0%B1%D0%BE%D1%80-%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0-%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0-%D0%B2-python-3b7e6cda5250>

Оставляю место.

**B9.6. Создание собственных структур**

место